Switch-Level Fault Simulation of MOS Digital Circuits

Michael Schuster
California Institute of Technology
Pasadena, California 91125
5132:TR:84

May 1, 1984

submitted to the Computer Science Department
in partial fulfillment of the requirements for the degree of Master of Science

Abstract

This thesis presents an algorithm for fault simulation of metal-oxide-semiconductor (MOS), field-effect transistor (FET) digital circuits. The circuits are modeled at the switch-level as networks of charge storage nodes connected with bidirectional transistor switches. Since the transistor structure of a MOS circuit is explicitly represented by its switch-level network, and since the circuit's logical behavior is modeled directly, the algorithm describes the behavior of defective MOS circuits with more accuracy than is possible with traditional logic gate fault simulation techniques. The algorithm is capable of analyzing a variety of MOS circuit defects, including the classical stuck-at-zero and stuck-at-one node faults, stuck-open and stuck-closed transistor faults, and resistive short and open faults in wires. By using the concurrent simulation technique, the algorithm requires far less computation than a simple serial simulation of each defective circuit.

Acknowledgements

I thank Dana Schuster for her continuous help and encouragement, Randy Bryant for his in-sights and algorithms that made this research possible, and Steve Jobs for the artifacts that eased the preparation of this thesis.

This research was supported in part by the IBM Corporation, in part by the Defense Advanced Research Contracts Agency, ARPA Order 3771, and in part by AT&T Bell Laboratories.

Table of Contents

Table of Figures

Introduction

As improvements in integrated circuit technology continue to increase circuit density and decrease circuit costs[1], test engineers are faced with the increasingly difficult problem of determining whether a circuit has been manufactured without defects[2]-[4]. The testing problem has two major components: *test generation* and *test verification* [5]. Test generation is the process of generating a sequence of test pattern stimuli that will demonstrate the circuit's correct operation. Test verification is the process of proving that a sequence of test patterns is effective.

Since a formal proof of the effectiveness of a test is impossible in practice[6], test engineers use fault simulation to determine how well a sequence of test patterns distinguishs a good circuit from a defective one. A fault simulator takes as input a description of the good circuit, a list of the observation points of the test (that is, the outputs of the circuit), a sequence of test patterns, and a set of hypothetical defects in the circuit, as shown in Figure 1. The simulator determines how the good circuit and all of the defective circuits behave by simulating the application of the test patterns to the inputs of the circuit.

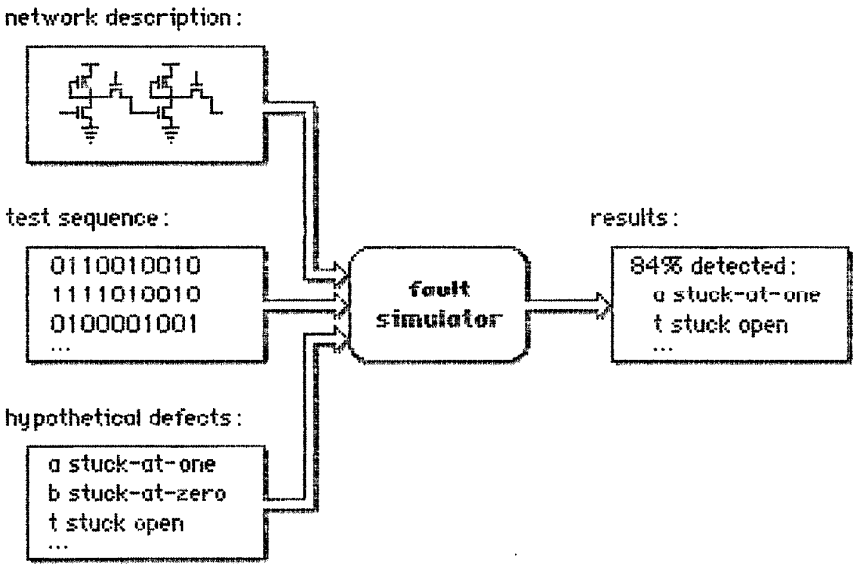

Figure 1. Inputs and outputs of a fault simulator

The test sequence is considered to have detected a fault when it causes the defective circuit to produce a value at some observation point that differs from the value produced by the good circuit. By keeping track of which faults have been detected and which have not, the simulator determines the *fault-coverage* of the test sequence, defined as the ratio of the number of

faults detected to the total number of faults simulated. The simulator also provides information about the operational characteristics of defective circuits that were not detected by the test sequence. Faults may not be detected because the test sequence failed to exercise the defective part of the circuit, or because the test sequence failed to make the results of such an exercise visible at an observation point. This information helps the test engineer modify or extend the test sequence to improve its fault coverage.

For a large digital circuit, thousands of hypothetical defects and thousands of test patterns must be simulated to adequately characterize the fault coverage of a test sequence. *Serial* simulation, in which the good circuit and each defective circuit are simulated separately, requires far too much computational effort. Fortunately, clever algorithms can reduce the amount of computation required considerably. Ulrich and Baker's *concurrent simulation* tech- nique[7] exploits the fact that the behavior of a defective circuit often differs only slightly from the behavior of the good circuit. Rather than simulating each defective circuit separately, a concurrent fault simulator selectively simulates only portions of the defective circuits to keep track of how their behaviors differ from the behavior of the good circuit. It appears as if the simulator simulates the good circuit and all of the defective circuits concurrently, but the amount of computation required is often only a small factor greater than that required to simulation the good circuit alone.

A concurrent fault simulator easily determines when a defective circuit produces a value that differs from the value produced by the good circuit without storing the entire output history of the good circuit simulation. As soon as the test sequence detects a fault, the simulation of the defective circuit is discontinued or *dropped*, thereby reducing the amount of computation required for the rest of the simulation. Typically, faults that cause large differences from the behavior of the good circuit, and hence require the most extra computational effort, are detected quickly. Consequently, fault dropping improves the overall performance of the simulator.

Most logic simulators model a digital circuit as a network of boolean logic gates connected by one-way, memory-less wires, in which each gate produces values on its outputs based on the values applied to its inputs, and possibly on the value of its internal state. These simulators are of limited value for determining the behavior of metal-oxide-semiconductor (MOS), field-effect transistor (FET) circuits[1]. They fail to capture properties such as the bidirectional nature of transistors, the ability of nodes to store charge, and the formation of logic levels by ratioed paths and charge sharing. As a result, many MOS circuit structures cannot be adequately modeled as a set of logic gates. Furthermore, fault simulators based on logic gates model only a limited class of faults, such as gate inputs and gate outputs stuck-at-zero or stuck-at-one. Faults such as open and short circuited transistors or opens and shorts in wires, which often change a combinational network into a sequential one, have no accurate and straightforward logic

gate equivalent[8],[9]. Hence, these faults are beyond the capabilities of most logic gate simulators.

To remedy these problems with conventional logic gate simulators, we use Bryant's *switch-level* logic model [10],[11] as a basis for our fault simulation algorithm. The switch-level model was developed to describe the logical behavior of MOS circuits at a level of detail which previously required accurate waveform simulation[12],[13], but with simulation times comparable to conventional logic gate simulators. In this model, a network consists of a set of nodes connected by transistor switches. Nodes have states 0, 1, and X representing low, high, and invalid (or uninitialized) voltages, respectively. Transistors have states 0, 1, and X representing open, closed, and indeterminate switches, respectively. Transistors have no assigned direction of information flow. They are assigned different strengths to model the effects of their relative resistance in ratioed circuits. Several types of transistors are provided to model both the nMOS and CMOS logic families. Nodes retain their states in absence of applied inputs, providing an idealized model of dynamic storage. They are assigned different sizes to model the effects of their relative capacitances in charge sharing.

Several logic simulators have been implemented based on switch-level models[14]-[16]. These programs have proved the switch-level approach valid for logic simulation because they efficiently model a large variety of MOS circuit structures. The switch-level model is suited for modeling defects in MOS circuits, because many important MOS circuit defects can be easily modeled by making smallmodifications to the switch-level representation of the good circuit. Hence, while the switch-level model has been successful for logic simulation, it seems especially attractive for fault simulation[17],[18]. Hayes[19] has proposed the connector-switch-attenuator representation of logic circuits for modeling faults, and the switch-level model has essentially the same capabilities.

We have adapted the technique of concurrent simulation to develop a switch-level fault simulation algorithm for MOS circuits, where the problem is viewed as one of simulating a large number of nearly identical switch-level networks. The algorithm accurately simulates a variety of MOS circuit structures, under a variety of fault conditions, at much higher speeds than is possible with serial simulation. Bose[20] implemented a concurrent fault simulator for MOS circuits, but this program models only a limited class of networks and faults. In this thesis, we present an overview of the switch-level model and show how to represent a variety of defects with it. We also present the concurrent simulation algorithm, show examples of its operation, and present experimental results obtained from an implementation.

Network Model

A switch-level network consists of a set of *nodes* connected by a set of *transistors*. The model places no restrictions on how the nodes and transistors are interconnected — any arbitrary network structure can be simulated. Each node has a state 0, 1, or X. States 0 and 1 represent low and high voltage levels, respectively. The X state represents an indeterminate voltage arising from an uninitialized node, from a short circuit, or from improper charge sharing.

Each node in a switch-level network is classified as either an *input* node or a *storage* node. An input node provides a strong signal to the network, as does a voltage source in an electrical circuit. The actions of the network do not affect the state of an input node. The power and ground nodes Vdd and Gnd are examples of input nodes. They act as constant sources of the states 1 and 0, respectively.

The operation of the network determines the states of the storage nodes. Each storage node holds its state when not connect to other nodes, as does a capacitor in an electrical circuit. To model circuits in which the relative capacitances of the nodes serve a logical function, each storage node is assigned a discrete *size* from the ordered set $\{k_1, k_2, \ldots, k_q\}$, where $k_1 < k_2 < \ldots < k_q$ A storage node with a larger size is assumed to have a much greater capacitance than a storage node with a smaller size. When a set of storage nodes share charge, the states of the largest nodes in the set override the states of the smaller nodes. The number of different sizes required in a switch-level network (q) depends on the circuit to be simulated. Most circuits can be represented with just two node sizes. In this representation, high capacitance nodes such as busses are assigned size k_2, and all other nodes are assigned size k_1. An input node is assigned size w to distinguish it from the storage nodes.

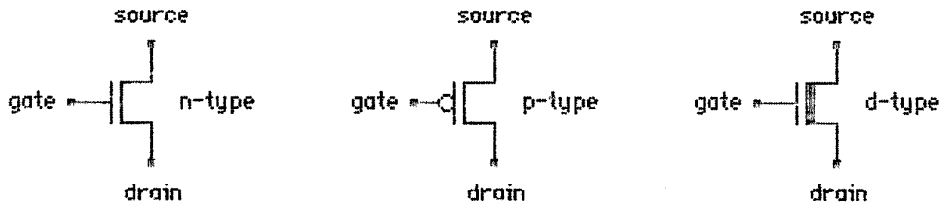

Figure 2. n-type, p-type, and d-type transistors

A transistor is a device with terminals labeled *gate*, *source*, and *drain*. The switch-level model makes no distinction between the source and drain terminals. Each transistor is symmetric and bidirectional. So that circuits in both the nMOS and CMOS logic families can be modeled, each transistor is classified either *n-type*, *p-type*, or *d-type*, as shown in Figure

2. The n-type and p-type transistors correspond to n-channel and p-channel devices, respectively. A d-type transistor corresponds to a negative threshold depletion mode device.

A transistor acts as a resistive switch connecting or disconnecting its source or drain nodes according to its type and the state of its gate node, as shown in Figure 3. The transistor states 0 and 1 represent nonconducting open and fully conducting closed conditions, respectively. The X state represents an indeterminate condition between open and closed, inclusive.

gate state	n-type	p-type	d-type
0	0	1	1
1	1	0	1
X	X	X	1

Figure 3. Transistor state function

To model circuits in which the relative resistances of transistors determines logical behavior, each transistor is assigned a discrete *strength* from the ordered set $\{g_1, g_2, \ldots, g_p\}$, where $g_1 < g_2 < \ldots < g_p$. A transistor with a larger strength is assumed to have a much greater conductance (that is, a much smaller resistance) than a transistor with a smaller strength. When paths of transistors in the 1 state connect a storage node to a set of input nodes, the state of the storage node depends only on the states of the input nodes connected to it by paths of largest strength. The strength of a path of transistors from an input node to a storage node equals the strength of the smallest transistor in the path. The number of different strengths required (p) depends on the circuit to be modeled. Most CMOS circuits do not utilize ratioed logic and hence can be modeled with just one transistor strength. Most nMOS circuits require only two strengths, with the d-type pull-up transistors assigned strength g_1 and all other transistors assigned strength g_2.

The switch-level model represents a MOS circuit in a highly idealized way. Only enough information about circuit parameters is included to describe the logical behavior of the circuit. As a result of this abstraction, the model may not predict the true behavior of circuits that rely on detailed analog properties such as sense amplifiers, arbiters, and delay sensitive circuits.

An example of a three transistor random access memory (RAM) circuit modeled at the switch-level is shown in Figure 4. The bus node is assigned size k_2 to indicate that it supplies its state to the selected size k_1 storage node (either m_1 or m_2) during a write operation (when either w_1 or w_2 has state 1) and to the size k_1 drain node of the selected storage transistor

(either c_1 or c_2) during a read operation (when either r_1 or r_2 has state 1). The d-type pull-up transistor is assigned strength g_1 to indicate that it drives the bus to 1 only when the strength g_2 pull-down transistor is not conducting. The strengths of all other transistors in the circuit are arbitrary, since they are not involved in ratioed path formation (except possibly when defects are present).

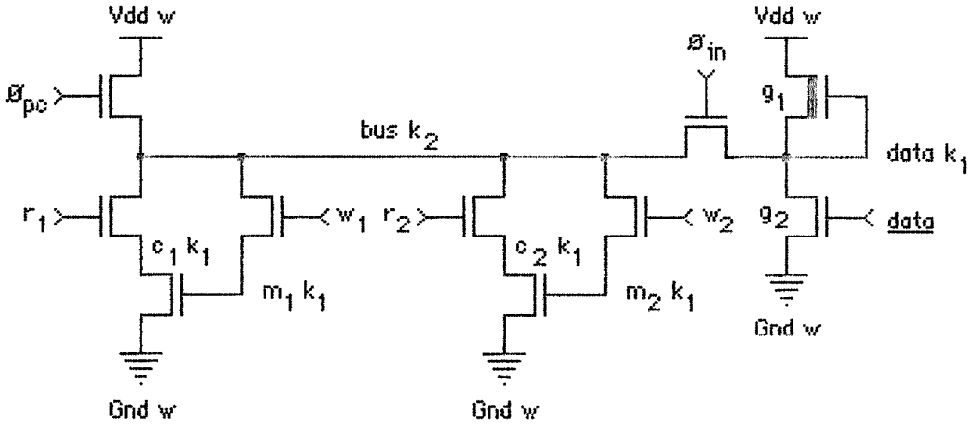

Figure 4. Three transistor dynamic RAM

An example of a CMOS dynamic shift register is shown in Figure 5. The inverters are implemented with complementary n-type pull-down and p-type pull-up transistors. All transistors are assigned strength g_1 since ratioed paths are not utilized. Both n-type and p-type transistors are used in parallel between inverters to avoid adverse threshold drops. The clock nodes ϕ_1 and ϕ_2 are non-overlapping. The nodes $\underline{\phi}_1$ and $\underline{\phi}_2$ are complements of ϕ_1 and ϕ_2, respectively. All storage nodes are assigned size k_1 since charge sharing is not utilized.

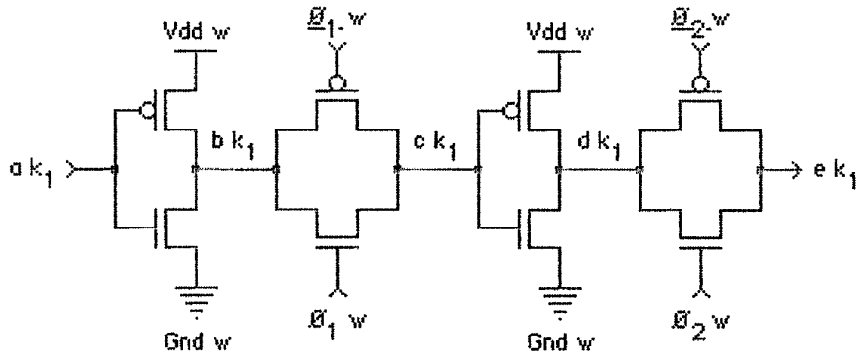

Figure 5. CMOS dynamic shift register

Fault Injection

Faults are injected by adding extra *fault transistors* to the switch-level description of the circuit, much like the scheme proposed by Lightner and Hachtel[21]. In the implementation, however, many faults are injected without actually adding fault transistors while providing behavior equivalent to what is described below. The gate nodes of these transistors are considered to be extra *fault inputs* to the network that control the presence or absence of the faults. A variety of MOS defects can be modeled with this method. For example, connecting two nodes with a fault transistor that is open in the good circuit and closed in the defective circuit generates a short circuit fault. Splitting a node into two parts and connecting the resulting nodes with a fault transistor that is closed in the good circuit and open in the defective circuit generates an open circuit fault. Since the state of each fault transistor is controlled independently by its fault input gate node, both single and multiple faults can be injected.

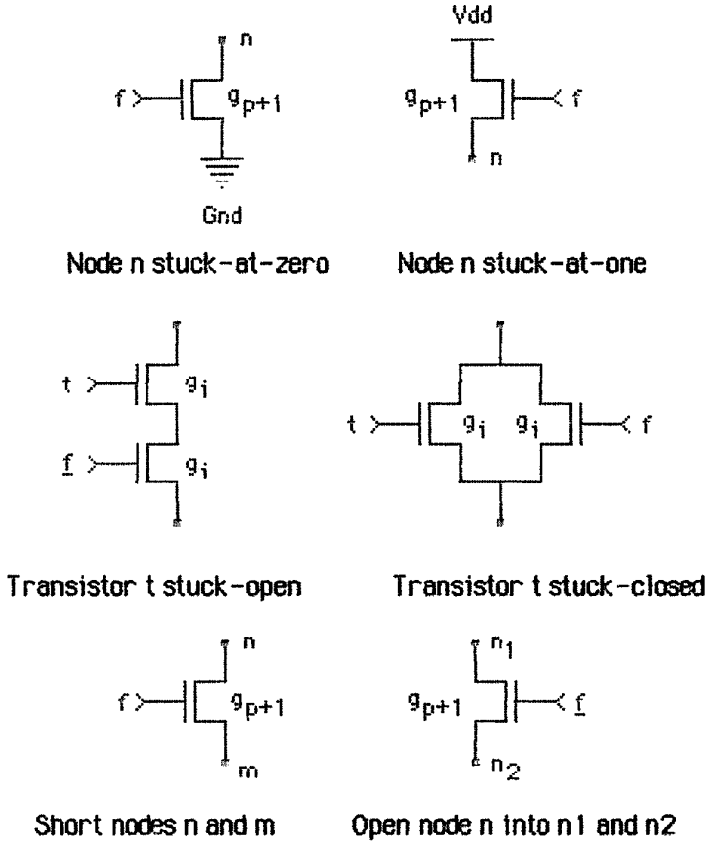

Figure 6. Modeling MOS defects

The strength of a fault transistor models the resistance of the short or open circuit in an approximate way. If the strength of the fault transistor is larger than the strengths of any other normal transistor in the network, then setting its state to 1 shorts its source and drain nodes

together so that they act as a single node. If its strength is equal to or smaller than other transistors, the presence of ratioed paths to its source or drain node may modify its effects.

Figure 6 illustrates the use of fault transistors to create a variety of defects. Those transistors with gate nodes labeled f are open in the good circuit and closed in the defective circuit; those with gate nodes labeled $\underline{f}$ are closed in the good circuit and open in the defective circuit. A stuck-at-zero or stuck-at-one node fault is modeled by inserting a fault transistor to short the node to Gnd or to Vdd, respectively. Its strength, g_{p+f}, is larger than other transistors in the network so that the node acts as Gnd or Vdd. A stuck-closed transistor fault is injected by shorting the transistor's source and drain nodes together with a fault transistor whose strength equals that of the defective transistor. A stuck-open transistor fault is modeled by putting a fault transistor in series with it. In our fault simulator, both stuck-at node faults and stuck-at transistor faults are implemented without extra fault transistors, while other faults require additional transistors to be inserted into the network.

Figure 7 shows the three transistor dynamic RAM circuit with two fault transistors added to inject the faults m_2 stuck-at-one and r_1 stuck-at-one. The fault m_2 stuck-at-one is injected by shorting the node to Vdd with a fault transistor whose gate node is labeled m_2@1. This node is set to 1 to inject the fault. The fault r_1 stuck-at-one is injected by shorting nodes bus and c_1 together with a fault transistor whose gate node is labeled r_1@1. We could have added a fault transistor to short r_1 to Vdd, however, we obtain the same effect by injecting a transistor stuck-closed fault. The node r_1@1 is set to 0 in the good circuit, and is set to 1 to inject the fault.

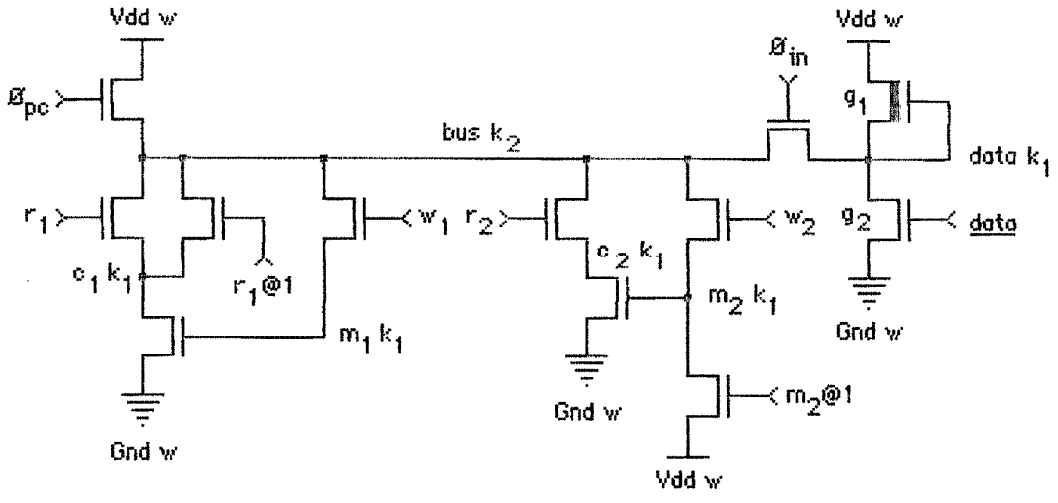

Figure 7. Injecting faults into the three transistor dynamic RAM

Although a variety of MOS defects can be injected by adding fault transistors, the switch-level model provides only a simplified representation of circuit failures. Subtle electrical defects, such as threshold drift, cannot be accurately described because the switch-level model captures only those aspects of a MOS circuit that determine its logical behavior, while abstracting away many of its detailed electrical properties. Defects affecting the delay in circuits whose behavior depends on the relative delay of several circuit elements cannot be described because the switch-level model uses an idealized timing scheme. The effects of resistive opens and shorts in wires can only be approximated because transistor strengths take on only discrete values. However, even if the fault models do not match the actual circuit failure modes, such models can still help in developing test patterns. For circuits implemented in bipolar technologies, experience has shown that a test sequence yielding high coverage of single stuck-at node faults in a logic gate network usually provides a good test of the circuit. We expect that the fault coverage measured by a switch-level fault simulator for some idealized set of faults reliably predicts how well the test sequence will work on a MOS circuit[8],[9],[22],[23].

Many faults create an X state on a node when in the good circuit the node has state 0 or 1. For example, if the control signal w_1 in the circuit shown in Figure 4 is stuck-at-zero, bit m_1 of the memory will never be initialized and will remain in the X state. On the other hand, if the precharge clock $\emptyset_{pc}$ is stuck-at-one, any time we attempt to read a 1 state out of a memory cell, a short circuit will develop between Vdd and Gnd producing an X state on the bus. Whether or not such X's would be detected in an actual test depends on detailed characteristics of the circuit that cannot be predicted at the switch-level, such as the initial voltages of dynamic nodes, how voltages would divide across a shorting path, and the thresholds of the devices sensing these X values. On one hand, an X state in a defective circuit should be considered undetectable, because there is no guarantee that the X will produce an effect different than the state of the node in the good circuit. On the other hand, a fault that prevents the circuit from being initialized, such as a stuck-at-zero clock line, would clearly be quickly detected. As a compromise, our fault simulator allows the user to specify a *soft detect* limit L such that if in the good circuit some observation point changes both to 1 and to 0 at least L times each, while the observation point in a defective circuit remains at X, then the fault is considered detected.

Behavioral Model

The switch-level model characterizes the behavior of a MOS circuit in terms of its *steady state response* function[10],[11], which can be explained informally by an analogy to electrical networks. A MOS transistor behaves as a voltage-controlled, non-linear resistive device where the voltages of its gate, source, and drain nodes control the resistance between its source and drain. Suppose the resistances of the transistors in a circuit could be controlled independently of their node voltages. For a given setting of transistor resistances, the circuit would act as a network of passive elements which, for a given set of initial node voltages, would have a unique set of steady state node voltages. Thus, a function mapping transistor resistances and initial node voltages to steady state node voltages gives a partial characterization of the behavior of a MOS circuit. The steady state response function provides just this sort of characterization, but in terms of the node and transistor states 0, 1, and X. That is, for a given set of initial node and transistor states, the steady state response function yields the set of states which the storage nodes would eventually reach if all the transistors were held fixed in their initial states. In this formulation, every storage node may represent a state variable because state can be stored as charge on storage nodes as well as in feedback paths. This function only approximates the behavior of the circuit since it does not describe the rate at which the nodes approach their steady states nor the effects of changing transistor states as their gate nodes change state.

```
begin
for each test pattern in the test sequence do
    begin
    set the input and storage nodes as specified by the test pattern;
    step := 0;
    while network is not stable and step < step limit do
        begin
        set transistor states according to their types and their gate node states;
        compute the steady state response for the storage nodes;
        set the storage nodes to their steady states;
        step := step + 1;
        end;
    end;
end;
```

Figure 8. Test sequence simulation algorithm

A switch-level network may contain nodes and transistors in the X state. Such states arise from improper charge sharing or from transient short circuits even in properly designed networks. The behavior of the network in the presence of X states must be described in a way that is neither overly optimistic (that is, ignoring possible error conditions), nor overly pessimistic (that is, spreading the X's beyond the region of indeterminate behavior). Bryant

accomplished this by carefully defining the steady state response function so that the steady state of a node is 0 or 1 if and only if it would have this unique state regardless of whether each node and transistor in the X state had state 0 or 1, otherwise the steady state of the node is X. The method used for computing the steady state response for an arbitrary set of node and transistor states achieves the same effect as computing the steady state response with nodes and transistors in the X state set to all possible combinations of 0 and 1, while avoiding the exponential complexity this process would entail.

Given a means for computing the steady state response function, a switch-level logic simulator simulates the operation of the circuit for each test pattern by repeatedly performing *unit steps* until a stable state is reached, as shown in Figure 8. We assume that the test patterns are applied slowly enough to the actual circuit for all nodes to reach stable states between each change in the test pattern inputs. Each unit step involves setting the transistors according to their types and the states of their gate nodes, computing the steady state response of the circuit, and setting the storage nodes to these values.

This simulation technique implements a *unit delay* timing model in which transistors switch one time unit (that is, one evaluation of the steady state response function) after their gate nodes change state. This scheme provides little information about the speed of the circuit but it usually sufficies to describe the circuit's logical behavior. As with other discrete time simulations, this computation may not reach a stable condition due to oscillations or perfectly matched delays. Hence, an upper bound, which varies with the size and the structure of the circuit, must be placed on the number of unit steps simulated.

Steady State Response

The steady state response function is defined formally in terms of a set of paths in a *switch graph* with vertices corresponding to the nodes in the network and edges corresponding to transistors in the 1 or X state. Each vertex is labeled with the size of its corresponding node. Each edge is labeled with the strength of its corresponding transistor. The source and drain nodes of transistors in the 0 state are considered to be electrically isolated since edges corresponding to these transistors are not included in the graph.

The effects of the initial state of one node on the steady state of another node through a series of transistors in the 1 or X state is described by the switch graph in terms of *rooted paths*. A rooted path consists of the vertex representing a root node, the vertex representing a destination node, and a set of edges representing the transistors that form a contiguous path from the root to the destination. A rooted path is said to be *definite* if and only if none of its edges correspond to transistors in the X state, since such a path will be present for any assignment of the states 0 and 1 to the transistors in the X state.

Each rooted path has a strength that equals the minimum of the size of its root and the strengths of the edges in its path, where sizes and strengths are ordered $\mu < k_1 < \ldots < k_q < g_1 < \ldots < g_p < w$. The strength of a rooted path approximates the amount of charge that would be supplied by the root vertex along the corresponding path in the circuit, and hence describes the path's relative importance in determining the steady state of the node represented by the destination vertex. The new value μ is introduced to conveniently represent the absence of a rooted path.

A rooted path of strength w is called an *input* path and consists of a single vertex corresponding to an input node and no edges. It represents a voltage source that supplies charge at an unbounded rate. A rooted path with strength in the set $\{g_1, \ldots, g_p\}$ is called a *driving* path and consists of a path from an input node through a non-empty set of transistors. It supplies charge at a rate limited by the path's conductance, which is characterized by the strength of the weakest transistor in the path. A rooted path with strength in the set $\{k_1, \ldots, k_q\}$ is called a *charging* path and consists of a path from a storage node through a (possibly empty) set of transistors. It supplies an amount of charge limited by the size of the storage node represented by the root vertex. A charging path with no edges represents the initial charge on the destination node itself.

To determine the net effect of a set of rooted paths to a vertex, two cases must be considered in which a path cannot represent a source of charge to the corresponding node in the circuit. First, any rooted path to a vertex with strength less than the strength of the strongest definite path to the vertex will be blocked because any charge that could be supplied along that path would be overwhelmed by the charge supplied by the stronger path, no matter whether transistors in

the X state had states 0 or 1. Second, even a path of maximum strength to a vertex may be blocked at some intermediate vertex because the charge that could be supplied along that path is overwhelmed by some other stronger definite path supplying charge to the intermediate vertex. Thus, a rooted path is said to be *blocked* if at some vertex along the path (possibly at its root or destination) there is a stronger *definite* rooted path to the vertex. That is, a source of charge will have no effect if and only if it is overridden at some node along the path for all possible assignments of the states 0 and 1 to the transistors in the X state.

Given the set of all unblocked paths to a particular vertex, the steady state response of the node corresponding to the vertex is determined by the initial states of the nodes corresponding to the root vertices of the paths. If none of the nodes corresponding to the root vertices have initial states 1 or X (respectively, 0 or X), then the steady state of the node corresponding to the destination vertex equals 0 (respectively, 1), otherwise its steady state equals X. That is, a node has steady state response equal to X unless the sources of charge represented by the unblocked paths to its corresponding vertex all act to drive or charge the node to 0 (or to 1).

This definition of the steady state response implies that the operation of the network will never change the state of an input node. This fact follows from the observation that the only unblocked path to an input node is the strength ω input path rooted at the node itself since all other paths to it have strength less than ω. Furthermore, this observation implies that sub-networks connected only through input nodes do not affect one another.

The formulation of the steady state response function in terms of the concise notions of rooted paths and path blocking provides a uniform and consistent way of determining the effects of nodes and transistors in the X state. Bryant showed that this formulation is correct in the sense that the steady state of a node is 0 or 1 if and only if it would have this unique state regardless of whether each node and transistor in the X state had state 0 or 1, otherwise the steady state of the node is X. Furthermore, since charging, driving, and input paths are all ranked in the same total ordering, this definition unifies the formation of logic levels by dynamic memory, charge sharing, and ratioed paths.

The transition from this graph formulation of the steady state response function to a simulation algorithm is accomplished in three steps. First, the strength of the strongest definite path to each vertex is found using a variant of Dijkstra's single source, shortest path algorithm[24]. Second, the strengths of the strongest unblocked path to each vertex from nodes in the 0 or X state and from nodes in the 1 or X state is found using another variant of Dijkstra's algorithm. In this step, the strength of the strongest definite path found in the first step imposes a threshold condition at each vertex that an unblocked path must satisfy. Finally, the strengths of these strongest unblocked paths determined in the second step are used to compute the steady state responses of the nodes.

Dijkstra's algorithm computes the lengths of the shortest paths from a distinguished source vertex to every other vertex in the graph, where the length of a path is defined as the sum of the non-negative, real valued edge weights. For our purposes, the "weight" of an edge is the strength of its corresponding transistor, the "length" of a path is the minimum of the strengths of its constituent edges, and the "shortest" path to a vertex corresponds to the strength of the strongest path to the vertex. In other words, we solve an isomorphic problem in which we find the path of least degradation (strongest) instead of the path of least length (shortest), where each additional edge in a path increases its degradation instead of increasing its length. Hence, if we think of a weaker path as being more degraded or longer, then it is easy to see that Dijkstra's algorithm solves this path problem[25],[26]. The requirement that edge weights be non-negative is satisfied since the strength of some initial part of a path is at least as large as the strength of the path itself.

Before Dijkstra's algorithm can be used to determine the strength of the strongest definite path to each vertex however, we must take into account the fact that the strength of a rooted path is the minimum of the size of the node corresponding to its root vertex as well as the strengths of its edges. This is accomplished by defining a *definite switch graph* with vertices corresponding to the nodes in the network and edges corresponding to the transistors in the 1 state. Edges corresponding to transistors in the X state are not included in this graph since we are interested only in definite rooted paths. Each edge has weight equal to the strength of its corresponding transistor. We add an artificial source vertex to the graph and connect it to every other vertex with an artificial edge that has weight equal to the size of node corresponding to the other vertex. Hence, a definite rooted path in the original switch graph is represented in the definite switch graph as a path from the source vertex through one edge to the root vertex and then through a (possibly empty) series of edges to the destination vertex. By taking the minimum of the edges weights of such a path, we obtain the strength of its corresponding definite rooted path.

Once the definite switch graph has been constructed, a modified version of Dijkstra's algorithm is applied to the graph to compute the strength of the strongest definite path to each vertex, as shown in Figure 9. V is the set of vertices in the definite switch graph and v0 is the artificial source vertex. E is the set of edges in the graph. The weight of an edge from vertex v to w is weight(v, w).

The correctness of this algorithm is proved by induction on the size of the set V - W that for each vertex v in V - W, q[v] is equal to the strength of the strongest definite path from v0 to v. Moreover, for each vertex w in W, q[w] is equal to the strength of the strongest definite path from v0 to w that lies wholly within V - W, except for w itself. Details of the proof can be found in Aho, Hopcroft, and Ullman[25]. When the algorithm terminates the set W is empty. Hence by the induction hypothesis, q[v] equals the strength of the strongest definite path to vertex v.

```
begin
for each v in V do
    q[v] := μ;
q[v0] := ∞;
W := V;
while W ≠ ∅ do
    begin
    choose w in W such that q[w] is maximized;
    delete w from W;
    for each (w, v) in E do
        q[v] := max(q[v], min(q[w], weight(w, v)));
    end;
end;
```

Figure 9. Dijkstra's algorithm for computing strength of strongest definite path

By representing the graph as an adjacency list and using a bucket sort to maintain a priority queue of the values q[v], the worst case time complexity of the algorithm is O(s + e), where s equals the total number of different node sizes and transistor strengths, and e equals the number of edges in the graph.

Once the strength of the strongest definite path to each vertex is found, we can determine the strengths of the strongest unblocked path to each vertex from nodes initially in the 0 or X state and from nodes initially in the 1 or X state. These strengths are found by using another variant of Dijkstra's algorithm.

The computation of the strengths of the strongest unblocked path from nodes initially in the 0 or X state to each vertex is accomplished by defining a *d switch graph* with vertices corresponding to the nodes in the network and edges corresponding to the transistors in the 1 or X state. Edges corresponding to transistors in the X state are included in this graph since we are interested in all potential paths. Each edge has weight equal to the strength of its corresponding transistor. Each vertex has a threshold equal to the strength of the strongest definite path to the vertex, so that we only consider unblocked paths with strengths that remain above these thresholds at each intermediate vertex. We add an artificial source vertex to the graph and connect it to every other vertex whose corresponding node has initial state 0 or X with an artificial edge whose weight equals the size of the node. Hence, an unblocked path from a node with initial state 0 or X is represented in the d switch graph as a path from the source vertex through one edge to the root vertex and then through a (possibly empty) series of edges to the destination vertex, such that the threshold requirement is satisfied at each intermediate vertex. By taking the minimum of the edge weights of such a path, we obtain the strength of its corresponding unblocked rooted path.

```
begin
for each v in V do
    d[v] := μ;
d[v0] := ω;
W := V;
while W ≠ ∅ do
    begin
    choose w in W such that d[w] is maximized;
    delete w from W;
    for each (w, v) in E do
        d[v] := max(d[v], limit(min(d[w], weight(w, v)), threshold(v)));
    end
end;
```

Figure 10. Computing strength of strongest unblocked paths from 0 or X nodes

Figure 10 shows a modified version of Dijkstra's algorithm that incorporates the threshold condition needed to compute d[v], the strength of the strongest unblocked path to each vertex v from nodes with initial state 0 or X. V is the set of vertices in the d switch graph and v0 is the artificial source vertex. E is the set of edges in the graph. The weight of an edge from vertex v to w is weight(v, w). The threshold of a vertex v is threshold(v) and equals the strength of the strongest definite path to v. The function **limit**(a, b) is used to impose the threshold condition. It equals a if $a \geq b$, otherwise it equals μ. Notice that the current strength of the strongest path to a vertex is updated only if a stronger path is found that satisfies the threshold condition. The correctness proof of this algorithm follows the proof of the algorithm shown in Figure 9.

The computation of u[v], the strength of the strongest unblocked path to each vertex v from nodes initially in the 1 or X state is accomplished by applying the algorithm shown in Figure 10 to a *u switch graph* that is similar to the d switch graph, except that the source vertex is connected to every other vertex whose corresponding node has initial state 1 or X with an artificial edge whose weight equals the size of the node.

Once we have determined the strengths of the strongest unblocked paths from nodes in the 0 or X state and in the 1 or X state, the steady state response of the nodes can be computed. If there exists no unblocked path to vertex v from a node initially in the 0 or X state, d[v] will equal μ, and the steady state of the node corresponding to v is 1. If there exists no unblocked path to vertex v from a node initially in the 1 or X state, u[v] will equal μ, and the steady state of the node corresponding to v is 0. Otherwise, neither d[v] nor u[v] will equal μ, indicating that there exists an unblocked path to v from a node in the 0 or X state as well as from a node in the 1 or X state. In this case, the steady state response of the node corresponding to v is X.

Example of a Steady State Computation

Suppose initially that the RAM circuit shown in Figure 4 has a steady state in which the nodes Vdd, bus, m_2, and data have state 1 and all other nodes have state 0. We will describe a write operation in which the nodes $\emptyset_{in}$ and w_2 are set to 1 so that a 0 state is written into the storage node m_2. Figure 11 shows the connected component of the circuit's switch graph containing the nodes of interest, that is, data, bus, m_2, Vdd and Gnd. Each vertex corresponds to a node and is labeled with its size. Each edge corresponds to a closed transistor and is labeled with its strength. We have set the strength of the transistors with gate nodes $\emptyset_{in}$ and w_2 to g_2 and g_1, respectively.

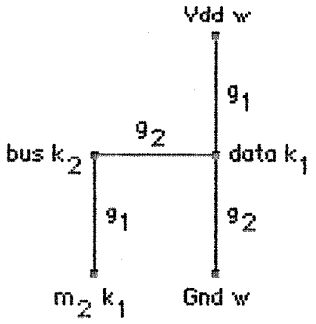

Figure 11. Switch graph of RAM circuit

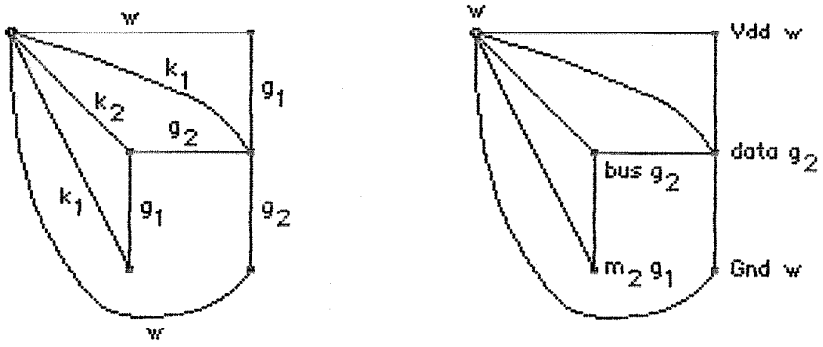

Figure 12. Definite switch graph

To find the strength of the strongest definite path to each vertex, we construct the definite switch graph shown at the left in Figure 12. We have connected the source vertex to every other vertex with an edge that has weight equal to the size of the node corresponding to the other vertex. The other edges correspond to closed transistors and have weight equal to their strength. The graph at the right in Figure 12 shows the strength of the strongest paths to each vertex. The

strongest paths from the source vertex to both data and bus pass through Gnd and not Vdd since the edge connecting data to Gnd is stronger that the edge connecting data to Vdd. On the other hand, the paths to m_2 passing through Vdd and Gnd have the same strength, since the strength of these paths is limited by edge connecting bus and m_2. The strength of the strongest path to the input nodes Vdd and Gnd is w.

To find the strength of the strongest unblocked path to each vertex from a node with initial state 0 or X, we construct the d switch graph shown at the left in Figure 13. We have connected the source vertex to each of the two nodes that have initial states of 0, that is, to data and Gnd, with an edge that has weight equal to the size of the node. Each vertex is labeled with a threshold equal to the strength of the strongest definite path to the vertex obtained from Figure 12. The path of strength k_1 from the source to data is blocked since it fails to satisfy the g_2 threshold condition at data. This path represents the initial charge of data that will be removed by the current to Gnd. The path of strength g_1 from the source through Gnd and data to Vdd is blocked since it fails to satisfy the w threshold condition at Vdd. This path represents a current that will be overwhelmed by the voltage source at Vdd. The paths from the source through Gnd to data, bus, and m_2 are not blocked. Hence, unblocked paths from nodes in the 0 state exist to the nodes Gnd, data, bus, and m_2, but not to Vdd.

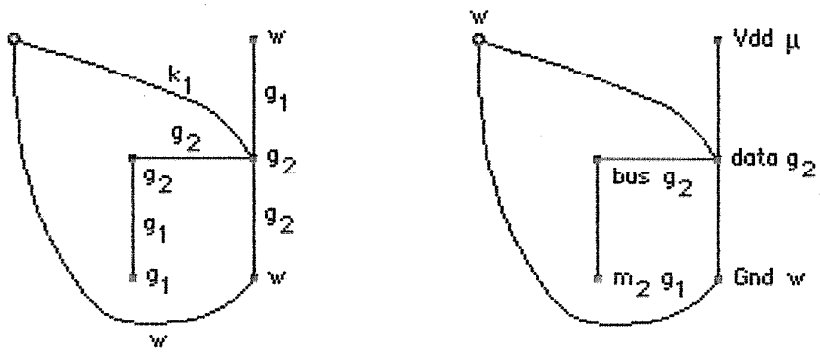

Figure 13. D switch graph

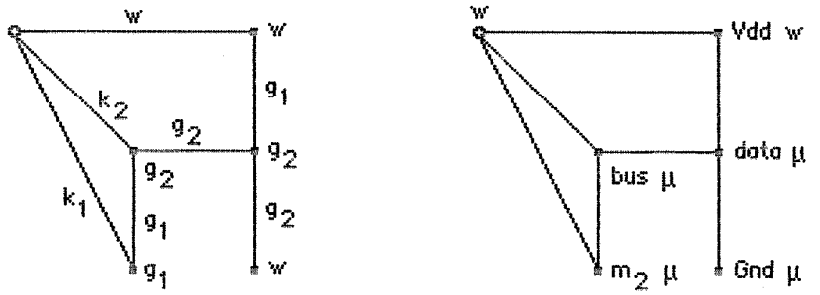

Figure 14. U switch graph

To find the strength of the strongest unblocked path to each vertex from a node with initial state 1 or X, we construct the u switch graph shown at the left in Figure 14. We have connected the source vertex to each of the nodes that have initial states of 1 with an edge that has weight equal to the size of the node. Each vertex is labeled with a threshold equal to the strength of the strongest definite path to the vertex obtained from Figure 12. The paths of strength k_1 and k_2 from the source to m_2 and from the source to bus are blocked since they fail to satisfy the threshold conditions at m_2 and bus, respectively. The path of strength g_1 from the source to Vdd to data is blocked since it fails to satisfy the g_2 threshold condition at data. Hence, unblocked paths from nodes in the 1 state exist only to Vdd.

The steady state response of the node can now be determined. Since there exist unblocked paths from nodes initially in the 1 state to Vdd, but none from nodes initially in the 0 or X state, the steady state response of Vdd is 1. Similarly, since there exist unblocked paths only from nodes initially in the 0 state to Gnd, data, bus, and m2, the steady state response of these nodes is 0.

Optimized Steady State Computation*

If the switch graph satisfies certain conditions, then the steady state response of a set of nodes can be determined with a technique that avoids the construction of the definite, d, and u switch graphs and the application of Dijkstra's algorithm. Consider a subcomponent of the switch graph in which each pair of vertices is connected by at least one path that does not pass through any intermediate vertices corresponding to input nodes. Suppose that no edge which connects vertices corresponding to two storage nodes represents a transistor in the X state. Furthermore, suppose that each of these edges has strength greater than or equal to the strength of each edge which connects a vertex corresponding to a storage node to a vertex corresponding to an input node. An example of a switch graph that satisifies the first of these conditions, but not the second, is shown in Figure 11. If the strength of the edge connecting bus and m_2 had strength at least as large as the strength of the edge connecting data and Gnd, then this graph would also satisfy the second requirement.

Although these conditions seem rather restrictive, many circuits encountered in practice meet these requirements once they are initialized so that most nodes and transistors have states 0 or 1. CMOS circuits that do not employ ratioed logic satisfy these conditions since they are modeled with just one transistor strength. All nMOS circuits with d-type pull-up transistors assigned strength g_1 and other transistors assigned strength g_2 satisfy these requirements. Furthermore, any subcircuit consisting of a set of storage nodes isolated from input nodes by open transistors also satisfies these requirements.

If the switch graph meets these conditions, then the strength of each path to a storage node rooted at an input node equals the strength of the first transistor in the path. This follows since all transistors connecting storage nodes are at least as strong as the transistors connecting storage nodes and input nodes. Hence, if the switch graph contains one or more vertices corresponding to input nodes, then the strength of the strongest definite path to every storage node equals the strength of the strongest transistor in the 1 state connecting an input node to a storage node. Otherwise, if the graph contains vertices corresponding only to storage nodes or if all input nodes are connected to the storage nodes by edges corresponding to transistors in the X state, then the strength of the strongest definite path to every storage node equals the size of the largest storage node in the graph. Thus, the strength of the strongest definite path to each storage node can be determined without applying Dijkstra's algorithm to the definite switch graph. We need only determine the strength of the weakest transistor connecting two storage nodes, the strength of the strongest transistor connecting an input node and a storage node, and

---

* This section may be skipped in the first reading of this thesis, since the material presented is not required for an understanding of our fault simulation algorithm.

the size of the largest storage node. If the strength of the edge connecting bus and m_2 in Figure 11 had strength g_2, then the strength of the strongest definite path to m_2, bus, and data would be equal to the strength of the transistor connecting data and Gnd, that is, g_2.

```
begin
s := w; q := d := u := μ;
for each v in V do
    begin
    if v corresponds to a storage node then
        begin
        q := max(q, size(v));
        d := max(d, low(state(v), size(v)));
        u := max(u, high(state(v), size(v)));
        end;
    end;
for each (v, w) in E do
    begin
    if v and w correspond to storage nodes then
        s := min(s, one(state((v, w)), strength((v, w))));
    else if v corresponds to a storage node and w corresponds to an input node then
        begin
        q := max(q, one(state((v, w)), strength((v, w))));
        d := max(d, low(state(w), strength((v, w))));
        u := max(u, high(state(w), strength((v, w))));
        end
    else if w corresponds to a storage node and v corresponds to an input node then
        begin
        q := max(q, one(state((v, w)), strength((v, w))));
        d := max(d, low(state(v), strength((v, w))));
        u := max(u, high(state(v), size((v, w))));
        end;
    end;
if s ≥ q and u < q then
    steady state response for all storage nodes in V is 0
else if s ≥ q and d < q then
    steady state response for all storage nodes in V is 1
else if s ≥ q then
    steady state response for all storage nodes in V is X
else
    optimized steady state computation can not be applied
end;
```

Figure 15. Optimized steady state computation algorithm

A similar technique can be used to determine the strengths of the strongest unblocked path to the vertices from nodes initially in the 0 or X state and from nodes initially in the 1 or X state. Since the transistors connecting storage nodes are at least as large as the transistors connecting input nodes to storage nodes, we can determine which rooted paths will be blocked by

simply checking the size of the root node and the strength of the first transistor in the path. A path to a storage node rooted at a storage node will be unblocked if and only if the size of its root is at least as large the strength of the strongest definite path to the root. A path to a storage node rooted at an input node will be unblocked if and only if the stength of the first transistor in the path is at least as large as the strength of the strongest definite path to the storage node. Hence, by keeping track of the size of the largest storage node in the 0 or X (1 or X) state and the strength of the strongest transistor connecting a storage node to an input node in the 0 or X (1 or X) state, the strength of the strongest unblocked path to each storage node from nodes initially in the 0 or X (1 or X) state can be easily determined.

Figure 15 shows the optimized steady state computation algorithm. The set of vertices V and the set of edges E form a subcomponent of the switch graph such that each pair of vertices is connected by at least one path that does not pass through any intermediate vertices corresponding to input nodes. For each vertex v in V, size(v) and state(v) equal the size and the state of the node corresponding to v, respectively. For each edge (v, w) in E, strength((v, w)) and state((v, w)) equal the strength and the state of the transistor corresponding to (v, w).

At the termination of the algorithm, the variable s equals the strength of the weakest transistor connecting two storage nodes, unless a transistor in the X state connecting two storage nodes is found. In this case, s equals μ. The variable q equals the size of the largest storage node, unless the graph contains one or more edges corresponding to transistors in the 1 state connecting a storage node and an input node. In this case, q equals the strength of the largest of these transistors. Hence, the optimized steady state computation technique can be applied to the graph if $s \geq q$. Furthermore, if this condition holds, then q equals the strength of the strongest definite path to each of the vertices corresponding to storage nodes. The function **one**(a, b) is used in the computation of s and q. It equals the strength b if the state a equals 1, otherwise it equals μ.

The variables d and u equal the maximum of the sizes of the storage nodes initially in the 0 or X (1 or X) state and the strengths of the transistors connecting storage nodes to input nodes initially in the 0 or X (1 or X) state. The function **low**(a, b) equals the strength b if the state a is 0 or X, otherwise, it equals μ. The function **high**(a, b) equals the strength b if the state a is 1 or X, otherwise, it equals μ. If $s \geq q$ and $d \geq q$ at the termination of the algorithm, then there exists an unblocked path to each storage node from a node initially in the 0 or X state. Similarly, if $s \geq q$ and $u \geq q$, then there exists an unblocked path to each storage node from a node initially in the 1 or X state. Thus, if $s \geq q$, then the values of d and u specify the steady state response of all of the storage nodes. If $s < q$, then this optimized computation technique can not be used, and the steady state responses must be found by the less efficient method of applying the variants of Dijkstra's algorithm to the definite, d, and u switch graphs.

Incremental Simulation

During the simulation of the network, often only a small portion of the network changes state on a given unit step, while the rest remains unchanged. Most logic simulators exploit this property, known as *latency*, by recomputing the output of a logic gate only if at least one of its inputs has changed state. We achieve a similar effect in switch-level networks by exploiting the *stability* of the steady state response function.

The steady state response function was described informally by an analogy to the formation of steady state node voltages in an electrical circuit composed of passive elements. From this analogy, we expect that once the storage nodes are set to their steady states, they will remain in these states until some transistor or input node changes state. That is, if the steady state response function is evaluated for a given set of initial node and transistor states and then re-evaluated using the steady states obtained from the first evaluation as the initial node states, the results of both of these evaluations should be identical. This property, known as the stability of the steady state response function, was proved by Bryant[11]. Stability indicates that the steady state response for some region of the network need not be recomputed until some input node or transistor in the region changes state. Hence, we can view network activity as creating small perturbations in the network, and compute the effects of these perturbations incrementally rather than recompute the state of the entire network.

We say that a storage node is *perturbed* if it is the source or drain of a transistor that has changed state, or if it is connected by a transistor in the 1 or X state to an input node that has changed state. Such a perturbation can only affect those storage nodes in the *vicinity* of the perturbed node. Two storage nodes are in the same vicinity if in the switch graph the vertices corresponding to these nodes are connected by a path which does not pass through any input nodes. A transistor in the 1 or X state is in the vicinity if its source or drain node is in the vicinity. An input node is in the vicinity if it is connected to a storage node in the vicinity by a transistor in the vicinity. This definition follows from the fact that the source and drain of a open transistor are considered to be electrically isolated, and from the observation that sub-networks connected only through input nodes do not affect one another. In most cases, a vicinity contains only several nodes and transistors, and hence activity in the network, for a given unit step, remains localized in small regions of the circuit.

The simulation algorithm shown in Figure 16 takes advantage of the fact that each unit step involves only incremental changes to the network state. Each test pattern consists of a set of node-state pairs of the form <n, v>, indicating a new state value v for the input or storage node n. For each pair specified, the routine *update_node* is called to update the state of the node. The routine *update_transistors* is then called to update the states of the transistors for which this node is the gate according to the table shown in Figure 3. Finally, the routines

perturb_node and *perturb_transistors* are called to accumulate the set P of storage nodes perturbed by these changes. A changing input node state perturbs all of the storage nodes connected to it by transistors in the 1 or X state, a changing storage node state perturbs that node alone, and a changing transistor state perturbs both its source and drain.

A series of unit steps is then performed, each simulating the effects of the perturbations of the storage nodes in P, until a stable state is reached or until a maximum step limit is exceeded. For each perturbed node in P, the routine *update_vicinity* is called to compute the steady state response of all storage nodes in the vicinity of the perturbed node. Those nodes which change state during this process are accumulated in the set Q. Once the effects of all perturbations have been simulated, the nodes in Q are set to their new states, the transistors whose gate nodes changed states are updated, and any nodes perturbed by the changing transistors are accumulated in a new set P in preparation for the next unit step. Note that setting a node to its steady state does not perturb it because, by the stability of the steady state response function, this state is stable. Hence, the routine *perturb_node* need not be called.

```
begin
for each test pattern in the test sequence do
    begin
    P := Ø;
    for each node-state pair <n, v> specified by the test pattern do
        begin
        update_node(n, v);
        update_transistors(n, v);
        P := P U perturb_node(n, v);
        P := P U perturb_transistors(n, v);
        end;
    step := 0;
    while P ≠ Ø and step < step limit do
        begin
        Q := Ø;
        for each node n in P such that not done(n) do
            Q := Q U update_vicinity(n);
        P := Ø;
        for each node-state pair <n, v> in Q do
            begin
            update_node(n, v);
            update_transistors(n, v);
            P := P U perturb_transistors(n, v);
            end;
        step := step + 1;
        end;
    end;
end;
```

Figure 16. Incremental test sequence simulation algorithm

Each vicinity whose steady state response is computed by *update_vicinity* may contain more that one perturbed node. In order to avoid duplicate steady state computations for these vicinities, we associate the flag done(n) for each perturbed node n. These flags are set to false by the routines *perturb_node* and *perturb_transistors*. The routine *update_vicinity* sets the flags of all storage nodes within the vicinity to true. Hence, *update_vicinity* need be called only for those perturbed nodes n in P such that done(n) is false.

The routine *update_vicinity* computes the steady state response for all storage nodes in the vicinity of a perturbed node. First, the nodes and transistors in the vicinity of the perturbed node are found by a variant of the *depth first search* algorithm[25]. Then the steady state responses for these nodes are computed either by using the optimized steady state response computation technique or by constructing the definite, d, and u switch graphs and applying the variants of Dijkstra's algorithm. Finally, the routine returns a set of node-state pairs indicating those nodes that changed state along with their new states.

```
begin
V := Ø;
E := Ø;
Q := {v0};
while Q ≠ Ø do
    begin
    delete v from Q;
    add v to V;
    if v corresponds to a storage node then
        begin
        for each edge (v,w) in the switch graph do
            begin
            if (v,w) not in E then
                add (v,w) to E;
            if w not in V and w not in Q then
                add w to Q;
            end;
        end;
    end;
end;
```

Figure 17. Search algorithm for finding the vicinity of a perturbed node

The search algorithm shown in Figure 17 is used by *update_vicinity* to find the connected subcomponent of the network's switch graph that contains the nodes and transistors in the vicinity of a perturbed node. In a manner similar to a depth first search of a undirected graph, it traces outward from the vertex corresponding to the perturbed node through edges corresponding to transistors in the 1 or X state until an input node is encountered. The vertex v0 corresponds to the perturbed node. Q contains the set of those vertices encountered in the search which are as yet unexplored. The set of vertices V and the set of edges E form the desired

connected subcomponent. We repeatedly select and delete an unexplored vertex v from Q and add it to V. If v corresponds to an input node, we need not trace any further, so we select another unexplored vertex v from Q. Otherwise v corresponds to a storage node, and we select an edge (v, w) incident upon v. This edge is added to E if it is not already in E. If w has not been explored, that is, it is not in V, then we add it to Q if it is not already in Q. Then we continue to select edges incident upon v until the list of these edges is exhausted. When Q is empty, V and E define the vicinity of the perturbed node v0.

Redundant Transistors*

Many circuits contain *redundant* transistors that, as a result of the current network state, temporarily cannot change the state of any node in the network. Consider the circuit shown in Figure 18. It consists of a two input *nor* gate in which the d-type pull-up has been replaced with a strength g_1 n-type transistor to make the example more interesting. A pass transistor with gate node $\emptyset_1$ connects node y, the output of the gate, to the node z. The states of the input nodes a and b have been set to 1 and 0, respectively, so that y has state 0. Assume that the initial states of the other nodes are as shown in the figure.

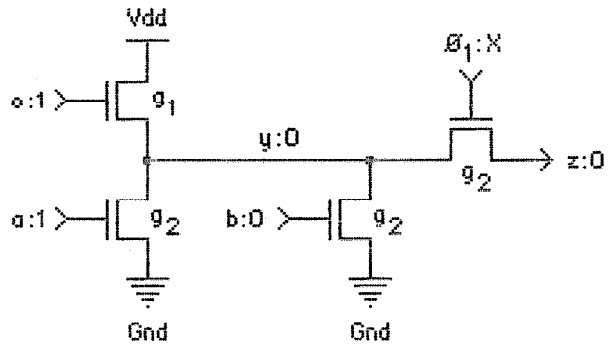

Figure 18. Circuit containing redundant transistors

Now suppose that node b changes to the state 1. Since y is already 0, this change has no effect on y nor on any other node in the circuit. In fact, changing the state of node b back to 0, or even to X, and then back to 1 an arbitrary number of times has no effect. This follows since y is the logical nor of the inputs a and b and the state of a is 1. Similarly, since the nodes y and z both have the same state, any change in the state of $\emptyset_1$ will have no effect on the states of y and z. If we view the circuit as a register that stores a value on node z, then it is easy to see that setting $\emptyset_1$ to 1, to write a 0 into the register that is already storing a 0, has no effect. Hence, both of these transistors are said to be *redundant*. Notice that the redundancy of a transistor depends upon both its initial state as well as the inital states of its source and drain nodes. If node a were to change to 0, y would change to 1 and these transistors would no longer be redundant.

In general, the switching of a transistor that is initially in the 0 or X state and whose source and drain nodes both have states 0 or 1 cannot change the network state. This observation can be motivated informally by the analogy between switch-level networks and electrical circuits composed of passive elements. If we repeatedly connect and disconnect two nodes in the circuit

* This section may be skipped in the first reading of this thesis, since the material presented is not required for an understanding of our fault simulation algorithm.

that initially have equal voltages with a resistor, then none of the voltages of any of the nodes in the circuit will change because there will be no current flow through the resistor. To show that such changes have no effect in a switch-level network, we can prove that all rooted paths passing through a redundant transistor satisfy two properties. First, any rooted path to some node through the redundant transistor is unblocked if and only if another rooted path to the node exists that does not pass through the transistor and whose root node has the same state as the root node of the path through the transistor. Second, if a path passing through a redundant transistor blocks a path that does not pass through the redundant transistor, then the states of the root nodes of the paths are equal. Hence, the presence or absence of rooted paths through a redundant transistor have no effect on the state of the network.

A transistor initially in the 1 state can also be redundant. Consider the pull-up transistor with gate node c. Even though the transistor has state 1, the states of its source and drain nodes are different. Hence, changing c to 0 or to X will have no effect on the network. In general, the switching of a transistor initially in the 1 or X state whose source and drain nodes have unequal states, neither of which is X, cannot change the state of the network. In analogy to an electrical circuit, two nodes connected by a resistor can have significantly different voltages only if the current supplied through the resistor is insignificant when compared with other current sources to the nodes. Hence, repeatedly removing and reinserting the resistor will have little effect on the voltages of the nodes. The fact that such changes have no effect in a switch-level network can be proved by showing that all rooted paths through the redundant transistor are blocked, and hence changing the state of the transistor does not affect the network state.

transistor in 0 state

drain \ source	0	1	X
0	1	0	0
1	0	1	0
X	0	0	0

transistor in 1 state

drain \ source	0	1	X
0	0	1	0
1	1	0	0
X	0	0	0

transistor in X state

drain \ source	0	1	X
0	1	1	0
1	1	1	0
X	0	0	0

Figure 19. Redundant transistor configurations

These observations are summarized in the tables shown in Figure 19. The 1 entries mark redundant configurations of transistor, source, and drain states. The 0 entries mark non-redundant configurations. These tables are used by the incremental simulation algorithm immediately after the steady state response is computed for a set of nodes in a vicinity. Each transistor with source or drain node in the vicinity that matches one of these configurations is marked redundant. All other transistors with source or drain in the vicinity are marked

non-redundant. These marks are checked on the next unit step when perturbations due to the changing transistor states are accumulated. Changing the state of a redundant transistor does not perturb its source nor drain. A redundant transistor may become non-redundant only when the steady state response is computed for its source or its drain as a result of a change in the state of some other non-redundant transistor.

Fault Simulation

As we have seen, the presence or absence of a fault in a switch-level network is controlled by the the state of a fault input node. Suppose the test patterns that specify clock and data values for the inputs of the circuit are extended so that they include values for the circuit's fault input nodes. To simulate the good circuit, the test patterns supply the appropriate values to the fault input nodes so that all faults are absent. To simulate a faulty circuit, the test patterns supply the same data and clock values, but the fault input nodes are set so that one or more faults are injected.

This scheme is illustrated in Figure 20. A nMOS dynamic shift register circuit is shown. It contains two fault transistors, one to inject the node b stuck-at-zero fault, and the other to short nodes b and c together. These fault transistors are controlled independently by the two fault input nodes named b@0 and b=c. Each test pattern specifies values for these nodes as well as values for the data input a and the clock inputs $\emptyset_1$ and $\emptyset_2$. Hence, the original inputs of the circuit are combined with the fault inputs to form an extended set of circuit inputs, each of which is supplied a value by the test patterns. The good circuit is simulated by setting the states of b@0 and b=c to 0 throughout the test sequence. The circuit containing the fault b stuck-at-zero is simulated by setting the state of b@0 to 1 and b=c to 0. The circuit containing both faults is simulated by setting the states of b@0 and b=c to 1.

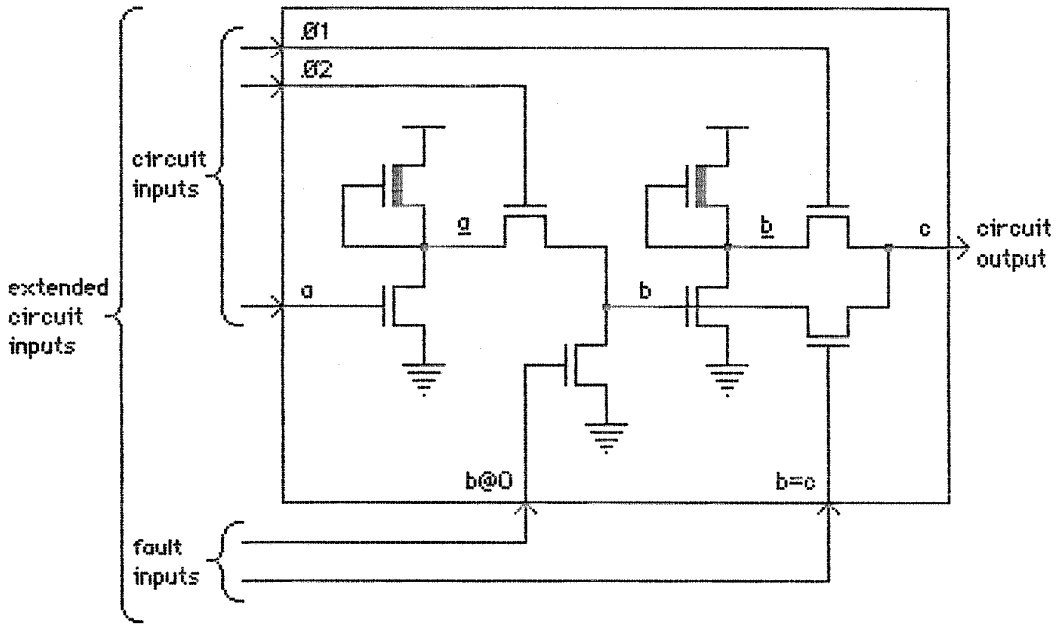

Figure 20. Extending circuit inputs with fault inputs

By including values for the fault input nodes as part of the test sequence, the behavior of the good circuit and a set of faulty circuits can be determined by repeatedly simulating test

sequences that differ only in selected fault input values. Hence, fault simulation can be viewed as the problem of efficiently applying a large number of nearly identical test sequences to a single circuit. This viewpoint separates the issues of fault modeling from fault simulation. Since values for the fault inputs are specified on an individual pattern-by-pattern basis, single, multiple, and even intermittent faults are easily modeled without changing the basic simulation algorithm. This scheme generalizes traditional fault simulation techniques which have the single stuck-at fault model built in to their representation of circuit structure[7],[20]. In fact, this scheme does not require that the clock and data inputs of all test sequences be identical. Thus, this technique is useful not only for fault simulation, but for multiple data simulation in which the clock and data values in each test sequence differ.

Our fault simulator determines the behavior of the good circuit and a set of m faulty circuits by applying a sequence of test patterns to a network that has been extended with a set of fault transistors and fault input nodes. Each test pattern in the sequence specifies values for the clock, data, and fault input nodes of the good circuit as well as for each of the faulty circuits. Each test pattern consists of a set of circuit-node-state triples of the form <c, n, v>, indicating a new state value v for the input or storage node n in circuit c, where $0 \leq c \leq m$. In this scheme, the good circuit is identified with the number 0 and the each faulty circuit is uniquely identified with the number i, where $1 \leq i \leq m$.

Since the values supplied by the test pattern to each of the faulty circuits differ from those supplied to the good circuit only in the values of the fault input nodes, we adopt the convention that each pattern explicitly specifies a value for a node in a faulty circuit only if its value differs from the value specified for the good circuit. This scheme avoids the unnecessary duplication of state information for each faulty circuit.

For example, suppose we wish to set nodes a and $\emptyset_1$ to 0 and $\emptyset_2$ to 1 in the circuit illustrated in Figure 20. In addition to the good circuit, we will be simulating three faulty circuits. The first, numbered 1, contains the node b stuck- at-zero fault. The second, numbered 2, contains a fault shorting nodes b and c together. The third, numbered 3, contains both of these faults. In this case, the test pattern consists of the set

{
<0,a,0>, <0,$\emptyset_1$,0>, <0,$\emptyset_2$,1>, <0,b@0,0>, <0,b=c,0>,
<1,b@0,1>,
<2,b=c,1>,
<3,b@0,1>, <3,b=c,1>
}

Thus, for the good circuit, values are specified for the fault input nodes b@0 and b=c as well as for the nodes a, $\emptyset_1$, and $\emptyset_2$. For faulty circuit 1, a value is specified only for the node b@0,

since the values for the other nodes are the same as those for the good circuit. Similarly, for faulty circuit 2, a value is specified only for the node b=c. Values for both of the nodes b@0 and b=c are specified for faulty circuit 3 since they both differ from the values specified for the good circuit.

This technique of explicitly representing a complete set of information only for the good circuit forms the foundation of our concurrent simulation algorithm. In addition to providing a concise representation for test patterns, this scheme provides a mechanism for compactly storing the state of all faulty circuits during the simulation as well as an algorithm for incrementally updating only those portions of the faulty circuits whose state differs from the state of those portions in the good circuit. The effectiveness of this technique rests on the assumption that the behavior of a faulty circuit is almost identical to the behavior of the good circuit. In practice, we have found this often holds.

Network State Representation

As the simulation proceeds, our fault simulator must maintain the current state of each node in the good circuit as well as in each faulty circuit. We exploit the property that the behavior of a faulty circuit often differs only slightly from the behavior of the good circuit by representing the state information for a node as a set of circuit-state pairs of the form <c,v>, indicating that the node has state v in circuit c, where $0 \leq c \leq m$. A pair corresponding to the state of a node in a faulty circuit is explicitly included in this list only when the state in the faulty circuit differs from the state of the node in the good circuit. In this case we say that the faulty circuit is *diverged* at the node, otherwise, the faulty circuit is said to be *converged* at the node.

The state information for each node is represented as a doubly linked circular *state list* of elements, to facilitate insertion and deletion. Associated with the node is a pointer to the first element in the list that specifies the state of the node in the good circuit. The elements corresponding to the diverged faulty circuits follow the first element in the list and are maintained in sorted order by their idenfication number.

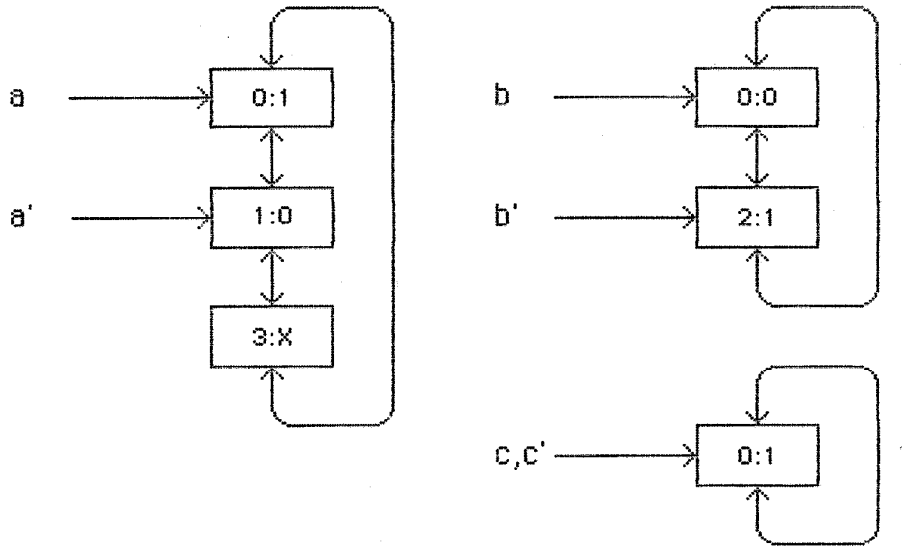

Figure 21. Network state representation

Figure 21 illustrates the state information associated with the nodes a, b, and c. An arrow labeled with the name of the node points to the first element in the list, which corresponds to the state of the node in the good circuit. Node a has state 1 in the good circuit, 0 in the faulty circuit numbered 1, 1 in faulty circuit 2, and X in faulty circuit 3. Hence, an element for faulty circuit 2 is not present in the list since its state is equal to the state in the good circuit. That is, faulty circuits 1 and 3 are diverged at node a, faulty circuit 2 is converged. Node b has state 0 in the good circuit as well as in the faulty circuits numbered 1 and 3. It has state 1 in faulty circuit 2. Node c has state 1 in the good circuit as well as in all faulty circuits.

A linear search of the list must be performed to determine the state of a node in a particular faulty circuit. If the element corresponding to the faulty circuit is not found in the list, then the state in the good circuit is used. To avoid repeatedly searching from the beginning of the list, a roving search pointer is maintained for each node. Hence, the search starts from the roving pointer and proceeds forwards or backwards in the list until the desired element is found or an element whose identification number is respectively greater than or less than the identification number of the desired faulty circuit. In either case, the roving pointer is left pointing to the last element referenced during the search. In Figure 21, an arrow labeled with the name of the node primed represents the element referenced by the roving pointer associated with the node.

```
begin
if c0 = 0 then
    rover := first
else if rover.c < c0 then
    begin
    rover := rover.next;
    while rover.c < c0 and rover ≠ first do
        rover := rover.next;
    end
else
    begin
    while rover.c > c0 do
        rover := rover.prior;
    end;
if rover.c = c0 then
    v0 := rover.v
else
    v0 := first.v
end;
```

Figure 22. Searching the state list of a node

Figure 22 shows the routine *find_element* which searches a node's state list to find its state in a particular circuit. C0 equals the identifying number of the desired circuit, such that $0 \leq c0 \leq m$. The variable first points to the element corresponding to the good circuit. The variable rover is the roving search pointer. Each element in the list has four fields, namely c, v, next, and prior. The c and v fields specify the number of the circuit corresponding to the element and the state of the node in this circuit, respectively. The next and prior fields point to the elements directly forwards and backwards in the list, respectively. When the routine terminates, v0 equals the state of the node in circuit c0.

When the state of a node in a faulty circuit is updated, an element may have to be added to or deleted from the state list of the node. If the faulty circuit is initially diverged and the new state is not equal to the state in the good circuit, then the value specified in the element is simply changed. If the faulty circuit is initially diverged and the new state of the node is equal to the node's state in the good circuit, then the element corresponding to the faulty circuit is deleted from the list. On the other hand, if the faulty circuit is initially converged and the new state is not equal to the state in the good circuit, then an element corresponding to the faulty circuit must be inserted into the appropriate position in the list.

```
begin
if c0 ≠ 0 then
    begin
    find_element(c0);
    if rover.c = c0 and first.v ≠ v0 then
        rover.v := v0
    else if rover.c = c0 and first.v = v0 then
        begin
        remove the element referenced by rover from the list;
        rover := first;
        end
    else if rover.c < c0 then
        begin
        insert an element just after the element referenced by rover;
        rover.next.c := c0;
        rover.next.v := v0;
        end
    else
        begin
        insert an element just before the element referenced by rover;
        rover.prior.c := c0;
        rover.prior.v := v0;
        end;
    end
else
    begin
    first.v := v0;
    rover := first.next;
    while rover ≠ first do
        begin
        if rover.v = first.v then
            remove the element referenced by rover from the list;
        rover := rover.next;
        end;
    end;
end;
```

Figure 23. Updating the state of a node

Updating the state of a node in the good circuit is more difficult. First, the whole state list associated with the node must be searched to find all of the faulty circuits that become converged as a result of this change. The elements corresponding to each of these faulty circuits must be deleted. A second difficulty results from the fact that a change in the state value of the element corresponding to the good circuit also changes the states of all currently converged faulty circuits. Hence, whenever the state of a node in the good circuit is updated, we may have to insert elements corresponding to each faulty circuit for which the node has not changed state and hence has become diverged. Since this set of faulty circuits is not represented in the state list, we must rely on our simulation algorithm to explicitly find this set.Figure 23 shows the routine *update_element* that changes the state of a node in the circuit c0 to v0. If c0 ≠ 0, then c0 identifies a faulty circuit. Hence, the routine *find_element* is called to adjust rover so that it refers to the element corresponding to c0, or to the element just before or just after the appropriate position in the list for c0. Then the element corresponding to c0 is updated, inserted into the list, or deleted from the list, as appropriate. On the other hand, if c0 = 0, then c0 identifies the good circuit. Hence, the state of the node in the good circuit is updated and a search is performed to find and remove all elements corresponding to faulty circuits that have become converged.

Concurrent Simulation Algorithm

The primary goal of our concurrent simulation algorithm is to exploit the fact that the behavior of a faulty circuit often differs only slightly from the behavior of the good circuit. We wish to extend our incremental simulation algorithm so that it selectively recomputes the steady state response only for those active regions of the faulty circuits that have behavior differing from the behavior of the good circuit.

Recall that the stability of the steady state response function indicated that the state of some region of the network need not be recomputed unless some input node or transistor in the region changes state. Hence, our incremental simulation algorithm needed to recompute states only for those storage nodes in the vicinity of a perturbed node. This technique can be extended to concurrent simulation by carefully keeping track of those circuits in which a storage node is perturbed.

First of all, we say that a storage node is *perturbed in the good circuit* if it is the source or drain of a transistor that has changed state in the good circuit, or if it is connected by a transistor in the 1 or X state in the good circuit to an input node that has changed state in the good circuit. This definition is the same as we previously stated, except that the references to the good circuit are made explicit. With this definition in mind, it would seem natural to say that a storage node is perturbed in a faulty circuit if it is the source or drain of a transistor that has changed state in the faulty circuit, or if it is connected by a transistor in the 1 or X state in the faulty circuit to an input node that has changed state in the faulty circuit.

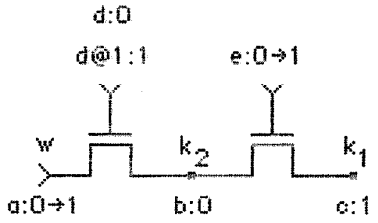

Figure 24. Perturbed nodes in a faulty circuit

This definition of a perturbed node in a faulty circuit, however, is unsatisfactory. Consider the circuit shown in Figure 24. Initially, the nodes a, b, and e have state 0 and node c has state 1. Furthermore, node d initially has state 0 in the good circuit and state 1 in the faulty circuit named d@1. Suppose that the input node a changes to 1. Since node d is 0 in the good circuit, this change does not affect nodes b and c. This is expected, since by our definition, neither of these nodes are perturbed in the good circuit. On the other hand, since node d has state 1 in the faulty circuit, this change causes node b to change to 1 in the faulty circuit. Hence, the definition given above of a node perturbed in a faulty circuit is inadequate since this change of node a in the good circuit affects the state of node b in the faulty circuit. In fact, this follows from the

observation that the change in the state of node a actually occurs in the faulty circuit as well as in the good circuit.

Now suppose that node e changes to 1. This change causes nodes b and c to be perturbed in the good circuit and hence the steady state response for both of these nodes is computed and found to be 0. On the other hand, since node d has state 1 in the faulty circuit and node a has state 1, this change causes both nodes b and c to change to 1 in the faulty circuit. Thus, we again see that our definition of a perturbed node in a faulty circuit is unsatisfactory since the change in node e would not have perturbed node b nor node c.

To avoid these problems, we say that a storage node is *perturbed in a faulty circuit* if it satisfies any one of the following conditions:

- The node is the source or drain of a transistor that has changed state in the faulty circuit.
- The node is connected by a transistor in the 1 or X state in the faulty circuit to an input node that has changed state in the faulty circuit.
- The node is connected by a transistor in the 1 or X state in the faulty circuit to an input node that has changed state in the good circuit.
- The node is contained within a vicinity in the good circuit of a node that is perturbed in the good circuit and, furthermore, the node is diverged in the faulty circuit.
- The node is contained within a vicinity in the good circuit of a node that is perturbed in the good circuit and, furthermore, the node is the source or drain of a transistor whose gate node is diverged in the faulty circuit.

This definition follows from our convention that a change of a node's state in the good circuit represents a change of the node's state in all converged faulty circuits. That is, when the state values in a faulty circuit for some vicinity are identical to those in the good circuit, then the steady state response computation for both of these circuits is identical, and hence, there is no need to duplicate the computation for the faulty circuit. On the other hand, when the state values in a faulty circuit differ from those in the good circuit, then a steady state response computation must be performed explicitly for the faulty circuit since the results may differ from the results for the good circuit. Thus, the steady state response for a region of the network is explicitly computed only for those active faulty circuits that have states which differ from the states in the good circuit.

This technique also solves the problem that arose from our state list representation of node states. Recall that when a change in the state of a node in the good circuit is recorded in the node's state list, the state of the node is effectively changed in all currently converged faulty

circuits as well as in the good circuit. Since we explicitly recompute states only for those differing faulty circuits, all faulty circuits in which a node becomes diverged will be found, while those faulty circuits whose state values are identical to the good circuit remain converged and are updated all at once when the change is recorded for the good circuit.

```
begin
for each circuit pattern in the circuit sequence do
    begin
    P := Ø;
    for each circuit-node-state triple <c, n, v> specified by the circuit pattern do
        begin
        update_node(c, n, v);
        update_transistors(c, n, v);
        P := P U perturb_node(c, n, v);
        P := P U perturb_transistors(c, n, v);
        end;
    step := 0;
    while P ≠ Ø and step < step limit do
        begin
        Q := Ø;
        for each circuit-node pair <c, n> in P in sorted order by c do
            begin
            Q := Q U update_vicinity(c, n);
            if c = 0 then
                P := P U perturb_vicinity(n);
            end;
        P := Ø;
        for each circuit-node-state triple <c, n, v> in Q in sorted order by c do
            begin
            update_node(c, n, v);
            update_transistors(c, n, v);
            P := P U perturb_transistors(c, n, v);
            end;
        for each observation node n do
            drop_diverged(n);
        step := step + 1;
        end;
    end;
end;
```

Figure 25. Concurrent fault simulation algorithm

Figure 25 shows the concurrent fault simulation algorithm that exploits the incremental nature of circuit activity and the similarity of behavior between the good and faulty circuits. As we previously mentioned, each test pattern consists of a set of circuit-node-state triples of the form <c, n, v>, indicating a new state value v for node n in circuit c, where $0 \leq c \leq m$ such that c = 0 identifies the good circuit and $c \neq 0$ identifies a faulty circuit. For each triple <c, n, v> specified by a test pattern, the routine *update_node* is called to update the state of node n to

the value v in circuit c using the algorithm shown in Figure 23. Then the routine *update_transistors* is called to update the states of the transistors for which this node is the gate according to the table shown in Figure 3. Finally, the routines *perturb_node* and *perturb_transistors* are called to accumulate the set P of storage nodes perturbed by these changes. The set P consists of circuit-node pairs of the form <c, n>, indicating that node n is perturbed in circuit c.

A series of unit steps is then performed, each simulating the effects of the perturbations in P, until a stable state is reached or until a maximum step limit is exceeded. For each perturbation <c, n> in P, the routine *update_vicinity* is called to compute the steady state response of all storage nodes in the vicinity of node n for the faulty circuit c. State values that are needed for this computation are found by searching the state lists associated with the node, using the algorithm shown in Figure 22. Those nodes which change state during this process are accumulated in the set Q of circuit-node-state triples <c, n, v>, each specifying a new value v for node n in circuit c.

For each perturbed node in the good circuit, the routine *perturb_vicinity* is called to find those nodes that are perturbed in faulty circuits as a result of conditions 4 and 5 described above. For each storage node n in the vicinity, perturbations <c, n> are added to P for every faulty circuit c that is diverged for node n. Furthermore, perturbations <c, n> are added to P for every faulty circuit c that is diverged for the gate node of a transistor whose source or drain is node n. These operations are performed by scanning through the state lists associated with the nodes. Notice that all steady state computations are performed before state changes for the good circuit are recorded. Hence, all computations for faulty circuits use the appropriate inital state values.

Once the effects of all perturbations have been simulated, the nodes in Q are set to their new states, the transistors whose gate nodes changed states are updated, and any nodes perturbed by the changing transistors are accumulated in a new set P in preparation for the next unit step. Elements from P and Q are processed in sorted order by the circuit number c to reduce the linear search time required by the algorithm shown in Figure 22. By using a bucket sorting technique, P and Q can be kept ordered in time linear in the number of faulty circuits being simulated.

After the nodes have been updated to their new states, the routine *drop_diverged* is called for each observation point in the circuit. Each faulty circuit diverged at an observation point, and hence detected by the test sequence, is dropped from the simulation. All state entries associated with the dropped circuits are deleted from the state lists of the nodes. Furthermore, all perturbations associated with these circuits are deleted from the set P.

Example

As an example of the use of a switch-level fault simulator, we will develop a set of test patterns for the nMOS pseudo-static register circuit shown in Figure 26. This circuit was chosen as an example because it illustrates many of the subtleties involved in testing MOS circuits. The register stores a value as dynamic charge on the size k_1 node a. Data is latched from the size k_2 bus node when the node write is set to 1. The bus is driven to 1 or 0 by setting the nodes up or down to 1, respectively. Within the register, the two inverters and the transistor whose gate node is refresh are used to amplify and feed back the latched data so as to counteract the decay of charge on node a. When data is to be transfered onto the bus from the register, the node read is set to 1. The value on the bus is amplified and inverted to form the node out, which is the output of the circuit.

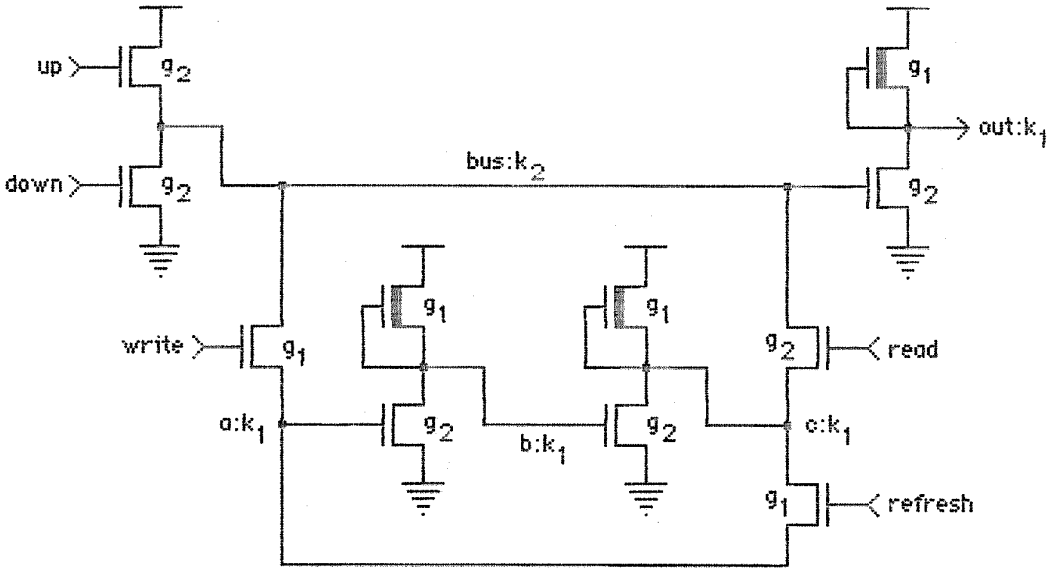

Figure 26. Pseudo-static register circuit

Our goal is to develop a sequence of test patterns that detect all possible single node stuck-at faults in the circuit. Some of these faults can be easily detected with a single test pattern. For example, the bus stuck-at-one fault is detected by setting the node down to 1 to drive the bus to 0. Hence, the node out will be 1 for the good circuit and 0 for the faulty circuit, since bus will stay at 1 in the faulty circuit. Other faults require a sequence of two test patterns to be detected. For example, to detect the node a stuck-at-zero fault, the first pattern latches a 1 into the register by setting the nodes up and write to 1 in the first pattern. Then, the second pattern drives the value in the register onto the bus by setting up to 0 and read to 1. Hence, out will be 0 for the good circuit and 1 for the faulty circuit.

Some of the node stuck-at faults require still more effort to detect. Consider the node up stuck-at-one fault. This fault will cause bus to stay in the state 1 unless the node down is set to 1, in which case a short circuit will develop between Vdd and Gnd producing X states on both bus and out. Whether or not the fault would be detected in the actual circuit depends on the voltage of bus as well as the threshold of out's pull-down transistor. In order to unambiguously detect the fault, down should be set back to 0 in the following test pattern. If the output of the circuit stays at 1, then the fault is not present, since bus is now isolated and will store its state of 0 as dynamic charge. On the other hand, if the fault is present, then bus will be driven to 1 and hence out will go to 0.

Consider the node write stuck-at-zero fault. This is one of the most difficult faults to detect, since node a will apparently remain in the X state and hence be undetectable. However, if we are careful, a value can be driven onto node a "backwards" by setting the nodes read and refresh to 1. To accomplish this, notice that node c, the output of the second inverter in the register, must also be driven to the same value. If we make the worst case assumption that node b is X, then the read transistor can be used to drive node c to 0, but not to 1. This follows from the observation that two unblock paths exist to node c that pass through the transistors of the inverter. The first is rooted at Vdd and has strength g_1. The second is rooted at Gnd and has strength g_2. If we try to drive c to 1 by setting the nodes up and read to 1, then an unblocked path rooted at Vdd of strength g_2 would be created, and hence c would be set to X. On the other hand, if we try to drive c to 0 by setting down and read to 1, then the path to c rooted at Vdd through the inverter will be blocked, and thus c will be set to 0. Therefore, by setting the nodes down, read, and refresh to 1, we can drive the nodes c and a to 0 independently of the initial states of a, b, and c.

pattern	up	down	read	write	refresh	out	faults detected
1	0	1	1	0	1	1	out@0, bus@1
2	1	0	0	1	0	0	out@1, bus@0, up@0
3	0	1	0	0	1	1	down@0
4	0	1	1	0	0	1	
5	0	0	0	0	0	1	up@1
6	0	0	1	0	0	0	read@0, read@1, write@0, write@1, down@1, refresh@1, a@0, b@1, c@0
7	0	1	1	0	1	1	
8	0	0	1	0	0	1	refresh@0, a@1, b@0, c@1

Figure 27. Test patterns for the pseudo-static register circuit

After node a is set to 0, a second test pattern changes the nodes up and write to 1 so that node a will be set to 1 in the good circuit. If write is stuck-at-zero, then node a will remain at 0 in the faulty circuit. The fault now behaves as the node a stuck-at-zero fault. Thus, a third pattern is used to drive the value in the register onto the bus by setting up to 0 and read to 1. Hence, out will be 0 for the good circuit and 1 for the faulty circuit.

Figure 27 shows a sequence of eight test patterns that detect all possible node stuck-at faults in the circuit. Each pattern consists of values for the input nodes up, down, read, write, and refresh. Also shown in the figure is the state of the node out for the good circuit after each test pattern has been applied. The rightmost column in the figure lists which faults are first detected by each pattern.

The basic strategy used by this test sequence is to first write a 1 into the register and then write a 0. Both values must be written to detect stuck-at faults on the nodes a, b, and c. Patterns 2 and 6 write and read a 1, respectively. Patterns 7 and 8 write and read a 0, respectively. Pattern 1, whose effect is described above, is used to detect the fault write stuck-at-0. After a 1 has been latched into the register by pattern 2, patterns 3 and 4 attempt to drive node a to 0 to detect the faults read stuck-at-one and refresh stuck-at-one. Since pattern 4 sets down to 1, pattern 5 can be used to detect the fault up stuck-at-one, as previously described.

	test pattern							
fault	1	2	3	4	5	6	7	8
good	0	1	1	0	1	1	0	0
out@0	0*	1	1	0	1	1	0	0
bus@1	X*	1	1	1	1	1	1	1
out@1	0	1*	1	0	1	1	0	0
bus@0	0	0*	0	0	0	0	0	0
up@0	0	0*	0	0	0	0	0	0
down@0	X	1	1*	1	1	1	1	1
up@1	X	1	1	X	1*	1	X	X
read@0	X	1	1	1	1	1*	1	1
read@1	0	X	0	0	0	0*	0	0
write@0	0	0	0	0	0	0*	0	0
write@1	0	1	X	0	0	0*	0	0
down@1	0	X	X	0	X	0*	0	0
refresh@1	0	X	X	0	0	0*	0	0
a@0	0	0	0	0	0	0*	0	0
b@1	0	0	0	0	0	0*	0	0
c@0	0	0	0	0	0	0*	0	0
refresh@0	0	1	1	0	1	1	0	1*
a@1	0	1	1	0	1	1	0	1*
b@0	0	1	1	0	1	1	0	1*
c@1	1	1	1	1	1	1	1	1*

Figure 28. States of node c for each faulty circuit and test pattern

Once the register has been read by pattern 6, pattern 7 is used to write a 0 into the register as well as detect the node refresh stuck-at-zero fault. This is accomplished by using a copy of pattern 1. If the fault is present, the node a will stay at 1. Pattern 8 performs the final read of the register and detects the remaining faults in the circuit.

More insight into the operation of the circuit in the presence of these faults can be obtained by studying Figure 28. It shows the state of node c for the good circuit and each faulty circuit after each test pattern has been applied. The starred entries indicate when the fault was first detected. As an exercise, determine why the fault down stuck-at-one is detected by pattern 6 even though node a is X. This exercise illustrates a common problem with a fault simulator. It provides information about the state of a faulty network, but not how it got there. On the other hand, through a complete understanding of the switch-level model, the simulator can provide invaluable information about the interplay between circuit defects, stored charge, and ratioed paths in a MOS circuit.

Performance

To evaluate the performance of our fault simulator FMOSSIM, we simulated two dynamic RAM circuits for different numbers of faults and different test sequences. All measurements were taken on a Digital Equipment VAX 11/780 running Berkeley 4.2bsd Unix for a version of the program written in the C language. The first circuit, a 64 bit memory named RAM64, contains 378 transistors and 229 nodes. The second, a 256 bit memory named RAM256, contains 1148 transistors and 695 nodes. The circuits incorporate a variety of MOS structures such as logic gates, bidirectional pass transistors, dynamic latches, precharged busses, and three transistor dynamic memory elements. We choose memory circuits to evaluate the simulator's performance because they could easily be scaled in size, and because they could be fully tested by test sequences consisting of special tests of the control and peripheral logic followed by a marching test[27] of the memory array. These circuits provide rather difficult test cases because the bit lines act as large global busses and hence activity is not well localized. When faults such as stuck-at-one control lines occur, locality is further reduced.

The circuits were simulated for randomly chosen subsets of the following fault classes: single storage nodes stuck-at-one, single storage nodes stuck-at-zero, and single pairs of adjacent bit lines shorted together. To validate the program, we also simulated other faults, including stuck-open and stuck-closed transistors. The performance characteristics for these faults did not differ significantly from those for node faults.

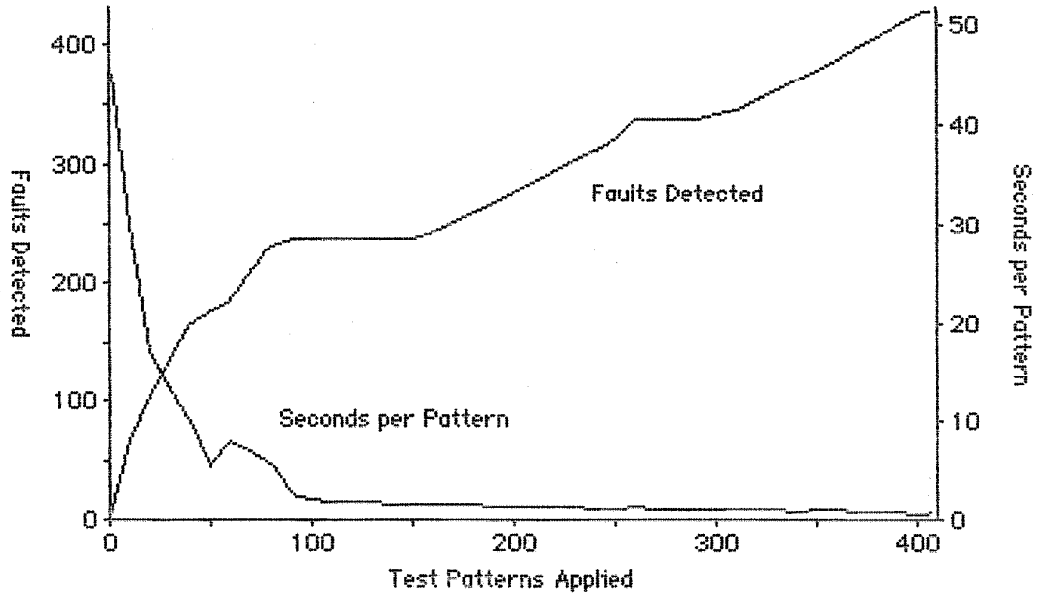

Figure 29. First test sequence for RAM64

Figure 29 illustrates the typical behavior of FMOSSIM when simulating a large number of faults. It shows the data for a simulation of RAM64 with 428 faults over a sequence of 407 patterns consisting of 7 patterns to test the control and peripheral logic, 40 patterns to perform a marching test of the row select logic, 40 patterns to perform a marching test of the column select and bit line logic, and 320 patterns to perform a marching test of the memory array. Each pattern actually represents a sequence of 6 input settings to cycle the circuit's clock nodes. Any time a faulty circuit produces a result on the output data node that differs from the result produced by the good circuit, the fault is considered detected, and the simulation of that circuit is dropped.

The rising curve in Figure 29 indicates the cumulative number of faults detected as the simulation proceeds. The falling curve indicates the CPU time required to simulate each pattern. This curve divides into two parts: the "head portion" consisting of the first 87 patterns during which all faults in the control and bus logic are detected, followed by the "tail portion" during which the faults in the memory array are detected. The simulation time starts at 45 seconds per pattern while the circuit is initialized and major faults such as stuck-at clock lines are being simulated. Those fault that create behavior vastly different from that of the good circuit, and hence require the most additional effort to simulate, are detected quickly. Once these faults are dropped, the performance of the simulator improves significantly. During the tail portion the simulator runs on the average just 3 times slower than it would to simulate the good circuit alone, even though as many as 190 circuits are being simulated simultaneously. The faulty circuits remaining during this portion behave much like the good circuit, because they contain only bit errors in the memory, which have no effect unless the faulty bit is selected for a read or write operation.

The entire fault simulation requires 21.9 minutes of CPU time, with 71% of the time consumed during the first 87 patterns. In contrast, the simulation of the good circuit alone requires 2.7 minutes, and a serial fault simulation in which each faulty circuit is simulated individually until it produces an output different from that produced by the good circuit would require an estimated 400 minutes (6.7 hours)*. This performance ratio of 18 for concurrent versus serial simulation is gained largely during the tail portion of the simulation, when many faults can be simulated concurrently at little additional cost.

Figure 30 illustrates how the choice of test sequence can affect the performance of the simulator. This simulation is the same as before, except that the patterns to test the row and column logic were omitted, leaving a total of 327 patterns. As a consequence, except for the 66

*All serial fault simulation times were estimated by summing over all faults the number of patterns required to detect the fault times the average time required to simulate the good circuit alone for 1 pattern.

faults detected during the first 7 patterns, all other faults are detected slowly as the marching test of the memory array proceeds, including faults in the address decoding and bus control logic. The time per pattern drops more slowly than before, because many faults that cause behavior much different from that of the good circuit remain undetected for a long time. This simulation required 49 minutes, even though the test sequence is shorter than before. Serial simulation, on the other hand, would require 450 minutes (7.5 hours), and hence concurrent simulation has a performance ration of only 9, due largely to the lack of a tail end effect. This result shows that the shortest test sequence for a set of faults may not give the shortest simulation time.

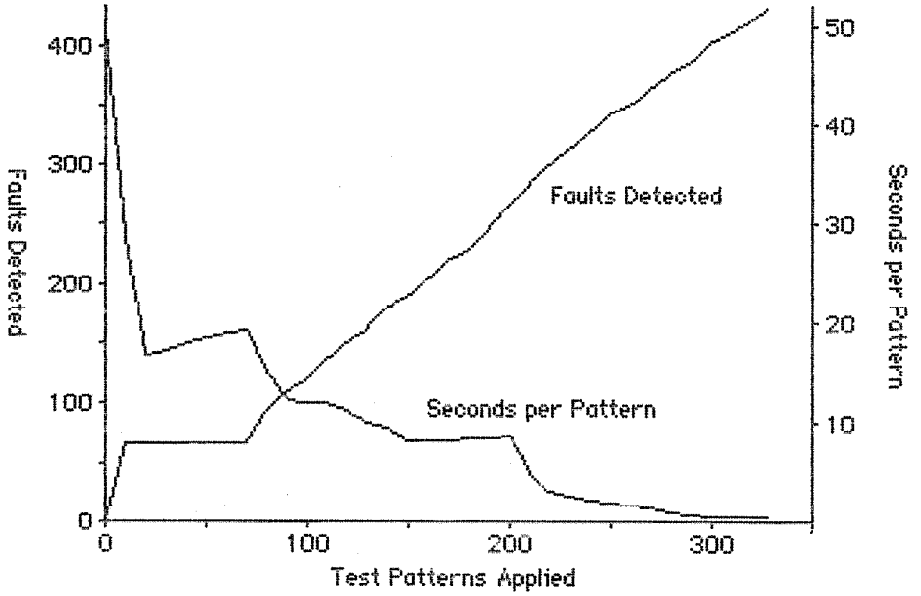

Figure 30. Second test sequence for RAM64

To see how the simulation time scales with the size of the circuit, we simulated RAM256 for a test sequence consisting of 1447 patterns similar to the first test sequence applied to RAM64. Simulating the good circuit alone requires 25.3 minutes. To run the test for all 1362 possible single stuck-at node and single bit line short faults, concurrent simulation requires 202 minutes (3.4 hours), while serial simulation would require an estimated 15,200 minutes (10.4 days!). Comparing these results to the time required for RAM64, we see that both the time to simulate the good circuit alone and the time for concurrent simulation has scaled up by a factor of 9, while the time for serial simulation has scaled by a factor of 37. This result makes concurrent simulation seem increasingly attractive as circuits grow large. It shows that concurrent simulation times for this class of circuits scales as the size of the circuit times the number of patterns, assuming the number of faults is proportional to the circuit size. Serial simulation time, on the other hand, scales as the product of all three factors.

Figure 31 illustrates the results of simulating RAM256 for different numbers of randomly selected faults, where the times shown are the average number of seconds per pattern over the entire length of the simulation. Note that the scale for serial simulation is 100 times that for concurrent. Both concurrent and serial simulation show a linear dependence on the number of faults, with serial being 85 times slower than concurrent. The linear growth of concurrent simulation shows that we pay no penalty for the overhead of maintaining lists of node states and determining the state of a node by searching the lists. On the other hand, it shows that our simulation algorithm exploits only the commonality between each faulty circuit and the good circuit. In many cases, two faulty circuits behave more nearly like each other than like the good circuit, although our algorithm does not exploit this fact.

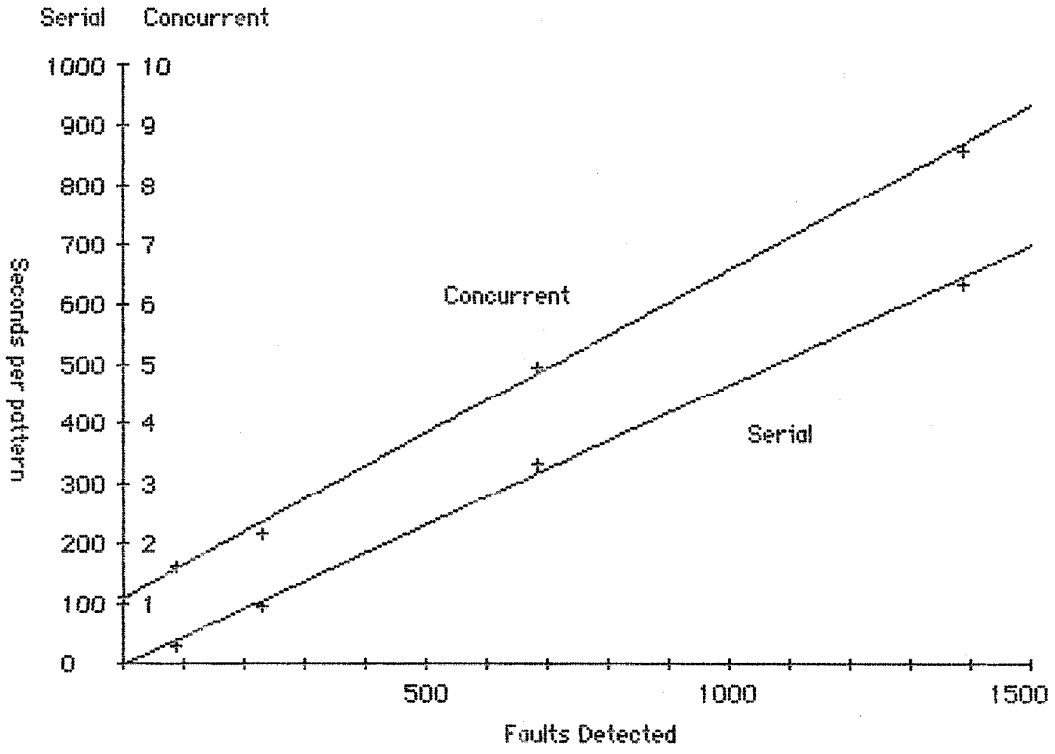

Figure 31. Average time per pattern versus number of faults for RAM256

Future Work

In the area of algorithms for concurrent fault simulation, further research is needed to develop techniques that exploit situations in which two faulty circuits are behaving more like each other than like the good circuit. We have experimented with a technique that discovers behavior common to two or more faulty circuits whenever the steady state response is computed for a vicinity in a faulty circuit. This technique, however, is not practical in most cases since its time complexity scales quadratically with the number of faulty circuits being simulated. A technique based on the Hopcroft's N log(N) set partitioning algorithm[25] might be applicable to this problem.

Much work remains in the area of test generation for MOS circuits. Faults such as stuck-open transistors which change a combinational network into a sequential one make this task formidable, especially since valid tests for such faults must take into account the possibility of sneak paths and critical races. We hope that the formalism provided by the switch-level model provides an adaquate basis for test generation.

We have made an initial step towards a test generation algorithm by developing a symbolic analysis algorithm which produces a set of expressions that describe when unblocked paths exist between any given root and destination node in a switch-level network[28]. Further work is needed to incorporate this algorithm into a practical test generation strategy. Of course, our fault simulation algorithm provides an experimental tool necessary for such research.

Summary

Our experience with the fault simulator has shown that it is a useful tool for developing test sequences. It provides information that is hard to obtain by any other means, even when developing a test for a small section of an circuit, such as an ALU or a register array. The simulator quickly directs the engineer to those areas of the circuit that require further tests. When developing the test sequences for the memory design described previously, we discovered that a simple marching test provided high fault coverage of the memory array itself, but that testing the control logic and peripheral circuits was more difficult.

Although it is difficult to characterize the performance of the algorithm for a wide variety of circuit structures, under a variety of fault conditions, we have found that it provides much higher performance than serial fault simulation. By using incremental computation techniques, the simulator updates only the active parts of the good circuit and the active parts of the faulty circuits that have behavior that differs from the behavior of the good circuit. By representing network state in a differential format, the simulator stores only the states of each node in the good circuit and the states of those nodes that have values for the faulty circuits that differ from their values for the good circuit. Furthermore, the simulator determines when a fault is detected without storing the entire output history of the good circuit simulation.

By basing the simulation algorithm on the switch-level logic model, we have developed a technique that that is well suited for modeling an important class of MOS circuit defects. Faults that affect the bidirectional nature of transistors, the ability of transistors to isolate stored charge on nodes, and the formation of logic levels by ratioed paths and charge sharing are easily modeled since these features of MOS circuits are modeled directly. Since faults are injected by making small modifications to the switch-level representation of the good circuit, single, multiple, and intermittent faults can be modeled without changing the basic simulation technique.

References

[1] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[2] H. Chang, E. Manning, and G. Metze, *Fault Diagnosis of Digital Systems*, Wiley-Interscience, NY, 1970.

[3] A. Friedman and P. Menon, *Fault Detection in Digital Circuits*, Prentice-Hall, NJ, 1971.

[4] M. Breuer, Ed., *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, CA, 1976.

[5] T. Williams, "Design for Testability - A Survey," *IEEE Transactions on Computers*, vol. C-31, no. 1, January 1982, pp. 2-15.

[6] O. Ibarra and S. Sahni, "Polynomially Complete Fault Detection Problems," *IEEE Trans- actions on Computers*, vol. C-24, no. 3, March 1975, pp. 242-249.

[7] E. Ulrich and T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," *IEEE Computer*, April 1974, pp. 39-44.

[8] K. Mei, "Bridging and stuck-at faults," *IEEE Transactions on Computers*, vol. C-23, no. 7, July 1974, pp. 720-727.

[9] R. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *Bell System Technical Journal*, Vol. 57, May-June 1978, pp. 1449-1473.

[10] R. Bryant, *A Switch-Level Simulation Model for Integrated Logic Circuits*, Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1981.

[11] R. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers*, vol. C-33, no. 2, February 1984, pp. 160-177.

[12] B. Chawla, H. Gummel, and P. Kozah, "MOTIS - a MOS Timing Simulator," *IEEE Transactions on Circuits and Systems*, vol. CAS-22, no. 12, December 1975, pp. 901-910.

[13] A. Vladimirescu, et. al., *SPICE Version 2G.5 User's Manual*, University of California, Berkeley, Technical Memo, August 1981.

[14] R. Bryant, "MOSSIM: A Switch-Level Simulator for MOS LSI," *18th Design Automation Conference Proceedings*, July 1981, pp. 786-790.

[15] C. Baker and C. Terman, "Tools for verifying integrated circuit designs," *Lambda Magazine*, 4th quarter 1980, pp. 22-30.

[16] R. Bryant, M. Schuster, and D. Whiting, *MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual*, Department of Computer Science, California Institute of Technology, 1982.

[17] M. Schuster and R. Bryant, "Concurrent Fault Simulation of MOS Digital Circuits," *Proceedings of the 1984 Conference on Advanced Reasearch in VLSI*, Massachusetts Institute of Technology, Cambridge, MA, 1984.

[18] R. Bryant and M. Schuster, "Performance Evaluation of FMOSSIM, a Concurrent Switch-Level Fault Simulator," unpublished paper.

[19] J. Hayes, "A Fault Simulation Methodology for VLSI," *19th Design Automation Conference Proceedings*, July 1982, pp. 393-399.

[20] A. Bose, et. al., "A Fault Simulator for MOS LSI Circuits," *19th Design Automation Con- ference Proceedings*, July 1982, pp. 400-409.

[21] M. Lightner and G. Hachtel, "Implication Algorithms for MOS Switch Level Functional Macro- modeling, Implication and Testing," *19th Design Automation Conference Proceedings*, July 1982, pp. 691-698.

[22] B. Courtois, "Analytical Testing of Data Processing Sections of Integrated CPUs," *1981 IEEE Test Conference Proceedings*, August 1981, pp. 21-28.

[23] J. Galiay, Y. Crouzet, and M. Vergniault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability," *IEEE Transactions on Computers*, vol. C-29, no. 6, June 1980.

[24] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Matematik* 1, 1959, pp. 269-271.

[25] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1975.

[26] R. Bryant, "Dijkstra's Algorithm — A Generalization," unpublished paper.

[27] S. Winegarden and D. Pannell, "Paragons for Memory Test," *1981 IEEE Test Conference*, pp. 44-48.

[28] M. Schuster, "Symbolic Analysis of Switch-Level Networks," unpublished memo.

Appendix – FMOSSIM Programmer's Manual

NAME

fmossim – switch-level fault simulator

SYNOPSIS

fmossim

DESCRIPTION

Fmossim is a fault simulator for metal-oxide-semiconductor (MOS), field-effect transistor (FET) digital circuits modeled at the switch-level. Fmossim determines how well a sequence of test patterns, when applied to the inputs of a circuit, distinguishes a good circuit from a defective one. It exploits the behavior common to the good and defective circuits to reduce the amount of computation required by the simulation.

Fmossim reads commands from the standard input file and writes the simulation dialog to the standard output file.

Fmossim takes a switch-level description of the circuit, a set of hypothetical faults, a set of observation points, and a sequence of test patterns. It determines how the good circuit and all faulty circuits behave by simulating the application of the test patterns to the inputs of the circuit.

A test sequence detects a fault when it causes the faulty circuit to produce at an observation point a value that differs from the value produced by the good circuit. Typically, Fmossim discontinues or drops the simulation of a faulty circuit as soon as the test sequence detects the fault.

NETWORK MODEL

A switch-level network consists of a set of nodes connected by idealized transistor switches. The model places no restrictions on the interconnections of the nodes and transistors.

Each node in a switch-level network has state 0, 1, or X. States 0 and 1 represent low and high voltages, respectively. The X state represents an indeterminate voltage arising from an uninitialized node, from a short circuit, or from improper charge sharing.

The switch-level model classifies each node as either input type or storage type. An input node provides a strong signal to the network, as does a voltage source in an electrical circuit. The actions of the network do not affect the state of an input node. The power and ground nodes Vdd and Gnd are examples of input nodes. They act as constant sources of the states 1 and 0, respectively.

The actions of the network determine the states of the storage nodes. A storage node holds its state when not connected to input nodes, as does a capacitor in an electrical circuit. To model the effects of charge sharing, the switch-level model assigns each storage node a positive integer size. A storage node with a larger size has a much greater capacitance than a storage node with a smaller size. When a set of storage nodes charge share, the states of the largest nodes in the set override the states of the smaller nodes.

A transistor is a device with terminals labeled gate, source, and drain. The switch-level model makes no distinction between the source and drain terminals. Each transistor is symmetric and bidirectional.

The switch-level model classifies each transistor as either n-type, p-type, or d-type. A d-type transistor corresponds to a negative threshold depletion mode device.

A transistor connects or disconnects its source and drain nodes according to its type and the state of its gate node, as shown below.

gate state	n-type	p-type	d-type
0	0	1	1
1	1	0	1
X	X	X	1

Transistor states 0 and 1 represent nonconducting open and conducting closed conditions, respectively. The X state represents an indeterminate condition between open and closed, inclusive.

To model ratioed circuits, the switch-level model assigns each transistor a positive integer strength. A transistor with a larger strength has a much greater conductance (that is, a much smaller resistance) than a transistor with a smaller strength. When paths of transistors in the 1 or X state connect a storage node to a set of input nodes, the state of the storage node depends only on the states of the input nodes connected by paths of largest strength. The strength of a path of transistors equals the strength of the smallest transistor in the path.

A switch-level network may contain nodes and transistors in the X state. Fmossim sets the state of a node to 0 or 1 if and only if the node would have this same state regardless of whether each node and transistor in the X state had state 0 or 1. Otherwise, Fmossim sets the node to X.

FAULT INJECTION

Forcing a node or transistor to a particular state generates a fault in a switch-level network. The actions of the network cannot alter the state of a forced node or transistor. Thus, a forced node acts like an input node. A forced transistor acts as if its gate node were *Vdd* or *Gnd*.

Forcing a node to 0 or 1 creates the classical stuck-at-zero or stuck-at-one faults, respectively. Forcing a transistor to 0 or 1 creates the stuck-open or stuck-closed faults, respectively.

Faults other than the stuck-at type, such as a short between two storage nodes, are generated by adding extra fault transistors to the network. The state of the fault transistor controls the presence or absence of the fault. Connecting two nodes with a fault transistor that is stuck-closed in the faulty circuit and stuck-open in the good circuit generates a short circuit fault. Splitting a node into two parts and connecting the resulting nodes with a fault transistor that is stuck-open in the faulty circuit and stuck-closed in the good circuit generates an open circuit fault.

The strength of a fault transistor models the resistance of a short or open circuit in an approximate way. If the strength of the fault transistor is larger than the strength of any other normal transistor in the network, then forcing its state to 1 shorts its source and drain together so that they act as a single node. If its strength is smaller than other transistors, the presence of ratioed paths to its source and drain may modify its effects.

The application of a test sequence causes each faulty circuit to behave differently. Fmossim keeps track of how the state of each faulty circuit differs from the state of the good circuit. The injection of a fault causes Fmossim to create a copy of the good circuit's network state. The copy diverges from the state of the good circuit as the test sequence exercises the defective part of the circuit. When the effects of these exercises propagate to an observation point, Fmossim drops the faulty circuit and deletes its network state.

Each faulty circuit is assigned a unique name. These names are used to distinguish between the good circuit and faulty circuits when the state of a node is observed or modified. Fmossim uses the syntax '*name*|*fault*:*state*' to indicate that the state of the node *name* for the faulty circuit *fault* is *state*. The vertical bar '|' separates the name of the node from the name of the faulty circuit. The colon ':' separates the name of the faulty circuit from the state of the node. The syntax '*name*:*state*' indicates that the state of *name* is *state* for the good circuit.

FAULT DETECTION

Often a fault creates an X state on a node when in the good circuit the node is 0 or 1. On one hand, the X is undetectable, since there is no guarantee that the X produces an observable effect. On the other hand, a fault that prevents the circuit from initializing, such as a stuck-at-zero clock line, is clearly detectable even though the circuit's observation points are always X. As a compromise, the user specifies a value L for the soft detect limit. Fmossim considers a fault to be detected if for the good circuit some observation point changes both to 1 and to 0 at least L times each, while remaining at X for the faulty circuit.

ENVIRONMENT

Fmossim maintains dictionaries of node, transistor, and faulty circuit names.

The **read** command obtains node names from the network description files. Transistors have names of the form '?n', where 'n' is a positive integer. Use the **status** command to find the names of transistors connected to a node of interest.

The **fault** command creates faulty circuits containing single node or transistor stuck-at faults. For example, the command 'fault a@0' creates the faulty circuit 'a@0' in which node 'a' is stuck-at-zero.

Combinations of the **test** and **force** commands create faulty circuits containing multiple node and transistor stuck-at faults. For example, the command 'test a@0b@1' followed by the command 'force a|a@0b@1:0 b|a@0b@1:1' creates faulty circuit 'a@0b@1' in which node 'a' is stuck-at-zero and node 'b' is stuck-at-one.

The **set** and **force** commands update node and transistor states to specified values. For example, the command 'set Vdd:1 Gnd:0' sets 'Vdd' to 1 and 'Gnd' to 0. The command 'force a|a@0:0' forces 'a' to 0 in faulty circuit 'a@0'.

The **simulate** command determines the effects of updated nodes and transistors on the network.

The **get** command lists node and transistor states for the good or faulty circuits. For example, the command 'get a b|a@0' lists the state of node 'a' for the good circuit and the state of node 'b' for the faulty circuit 'a@0'.

The **observe** command checks the states of specified observation points to find and drop detected faults.

COMMANDS

A command is an input line composed of a sequence of blank or tab separated words or tokens, the first of which specifies the command to be executed. The commands are:

quit

Causes the program to return to the shell.

help

Lists all commands along with a description of their use and function.

read { file }

Reads network descriptions from the specified files. Merges nodes in different files with the same name. Sets the states of all nodes to X. The default filename suffix is '.ntk'.

vector name { node | transistor | vector }

Declares a vector called name to represent a sequence of nodes and transistors. A vector of length one acts as a synonym for a node or transistor. A vector of length larger than one provides a convenient shorthand notation for updating and listing the states of a group of nodes or transistors.

initialize

Sets the states of all nodes in the network to X. Drops all faulty circuits.

option { option:value | limit:value }

Sets options and limits to specified values. The command with no arguments lists their values.

Setting the **debug** option non-zero lists program debugging information. Its initial value is zero.

Setting the **echo** option non-zero lists commands read from source files. Its initial value is zero.

Setting the **redundant** option non-zero identifies redundant transistors that temporarily cannot affect the state of the network. Its initial value is one.

Setting the **weaken** option non-zero causes a slightly less pessimistic X model to be used. Any transistor in the X state is treated as if it were slightly weaker than its nominal strength and hence can be overridden more easily. Its initial value is zero.

Setting the **case** option non-zero causes upper and lower case distinctions to be ignored when node, vector, and faulty circuit names are compared to entries in the dictionaries. Its initial value is zero.

The value of the **step** limit determines the maximum number of unit simulation steps that the **simulate** command performs before it forces the states of changing nodes to X. Its initial value is one hundred.

The values of the **soft** and **hard detect** limits determine when the **observe** command drops faults. Their initial values are zero.

dump { file }
Writes the state of the network at the completion of the most recent **simulate**, **initialize**, **read**, or **load** command to the specified file. The default filename suffix is '.dmp'.

load { file }
Reads the state of the network from the specified file create by the **dump** command. The default filename suffix is '.dmp'.

copy { file }
Writes a copy of the command dialog to the specified file. The command with no arguments stops the copying. The default filename suffix is '.cpy'.

source { file }
Reads commands from the specified files. The default filename suffix is '.src'.

comment { text }
Documents the simulation by listing the argument text.

test { name }
Creates names for faulty circuits. The command with no arguments lists the names of all faulty circuits.

untest { name }
Drops faulty circuits. The command with no arguments drops all faulty circuits.

set { node:state | node|fault:state | transistor:state | transistor|fault:state | vector:states | vector|fault:states }
Sets nodes and transistors to the specified states for the specified good or faulty circuits. Setting a vector is equivalent to setting each of its element nodes and transistors. The actions of the network may subsequently alter the state of a set node or transistor. To keep this from occurring, use the **force** command.

If the argument has the form 'node|fault:state', 'transistor|fault:state', or 'vector|fault:states', the command sets the state of the node, transistor, or vector to the specified value for the faulty circuit 'fault', respectively.

If the argument has the form 'node:state', 'transistor:state', or 'vector:states', the command sets the state of the node, transistor, or vector to the specified value for the good circuit, respectively. Further, if the node or transistor was forced in the good circuit, the command sets its state to the specified value for all faulty circuits. If the node or transistor was set in the good circuit, the command sets its state to the specified value in all faulty circuits except those for which it is forced.

force { node:state | node|fault:state | transistor:state | transistor|fault:state | vector:states | vector|fault:states }
Forces nodes and transistors to the specified states for the specified good or faulty circuits. Forcing a vector is equivalent to forcing each of its element nodes and transistors. The actions of the network cannot alter the state of a forced node or transistor. Only the **force** or **set** command will change its state.

If the argument has the form 'node|fault:state', 'transistor|fault:state', or 'vector|fault:states', the command forces the state of the node, transistor, or vector to the specified value for the faulty circuit 'fault', respectively.

If the argument has the form 'node|state', 'transistor|state', or 'vector|states', the command forces the state of the node, transistor, or vector to the specified value for the good circuit and all faulty circuits, respectively.

fault { node@state | transistor@state | vector@state | *@state | ?*@state }
Creates faulty circuits that contain single node or transistor stuck-at faults. Use the **test** and **force** commands to create faulty circuits containing multiple faults.

The command 'fault node@state' first creates faulty circuit 'node@state' via the 'test node@state' command. It then forces the state of 'node' to 'state' for this faulty circuit via the 'force node|node@state:state' command. Similarly, the command 'fault transistor@state' creates faulty circuit 'transistor@state' and then forces the state of 'transistor' to 'state' for this faulty circuit.

The command 'fault vector@state' performs the command 'fault element@state' for every element in the vector.

The commands 'fault *@state' and 'fault ?*@state' perform the commands 'fault node@state' and 'fault transistor@state' for every node and transistor in the network, respectively.

The command will not create a stuck-closed fault for a d-type transistor, since such a fault is meaningless.

unfault { node@state | transistor@state | vector@state | *@state | ?*@state }
Drops faulty circuits that contain single node or transistor stuck-at faults.

The commands 'unfault node@state' and 'unfault transistor@state' drop the faulty circuits 'node@state' and 'transistor@state', respectively.

The command 'unfault vector@state' drops the faulty circuits 'element@state' for every element in the vector.

The commands 'unfault *@state' and 'unfault ?*@state' drop the faulty circuits 'node@state' and 'transistor@state' for every node and transistor in the network, respectively.

get { node | node|fault | transistor | transistor|fault | vector | vector|fault }
Lists the states of the nodes, transistors, and vectors for the specified good or faulty circuits.

If the argument has the form 'node', 'transistor', or 'vector', the command lists the state of the node, transistor, or vector for the good circuit, respectively.

If the argument has the form 'node|fault', 'transistor|fault', or 'vector|fault', the command lists the state of the node, transistor, or vector for the faulty circuit 'fault', respectively.

observe { node | vector }

Checks the states of specified observation points to find and drop detected faults. Lists the observation points at which the fault is detected.

For each argument node, the command increments one of three integer variables associated with each faulty circuit 'fault' for which the node's state differs from its state for the good circuit. If the state of the node is 0 for the good circuit and 1 for 'fault', or 1 for the good circuit and 0 for 'fault', 'fault's' hard-detect variable is incremented. If the state of the node is 0 for the good circuit and X for 'fault', 'fault's' soft-0-detect variable is incremented. If the state of the node is 1 for the good circuit and X for 'fault', 'fault's' soft-1-detect variable is incremented.

The command drops 'fault' if the following condition holds between 'fault's' three variables and the **soft** and **hard detect** limits (set by the **option** command): hard-detect ≥ **hard** limit > 0 or (soft-0-detect ≥ **soft** limit > 0 and soft-1-detect ≥ **soft** limit > 0).

simulate

Performs unit simulation steps until the network is stable. The value of the step limit (set by the **option** command) determines the maximum number of unit simulation steps that the command performs before it forces the states of the changing nodes to X.

Each unit step corresponds to the propagation of a state change through one level of transistor logic. Typically, an oscillation induced by a fault causes the simulation to exceed the step limit. Forcing the states of the changing nodes to X terminates the oscillation.

status { node | transistor | vector | fault }

Lists status information about nodes, transistors, vectors, and faulty circuits.

Node status includes its size, its state, and a list of all transistors connected to it.

Transistor status includes its type, its strength, its state, and the names and states of its gate, source, and drain nodes.

Vector status includes information for each of its element nodes and transistors.

Faulty circuit status includes the values of its soft-0-detect, soft-1-detect, and hard-detect variables. Also included is a list of those nodes whose state for the faulty circuit differs from their state for the good circuit.

statistics

Lists simulation statistics gathered since the last statistics command. Statistics include the accumulated cpu time, the number of unit simulation steps performed, the number of nodes that have changed state, the number of faulty circuits dropped, and the number of vicinities simulated. A vicinity is a set of storage nodes connected by transistors in the 1 or X state.

The statistic 'num%n' specifies that only 'num' percent of the nodes in simulated vicinities changed state. The statistic 'num%s' specifies that the optimized steady state computation was valid for 'num' percent of the vicinities simulated. The statistics 'num%o' specifies that no nodes changed state for 'num' percent of the vicinities simulated.

NETWORK DESCRIPTION

The description of a switch-level network consists of a sequence of statements, each terminated by a semicolon ';'. A statement is composed of a sequences of words, numbers, and tokens, each separated by at least one blank, tab, or new-line character. Statements are distinguished by their first token. The statements are:

i name ;
Creates an input node with the specified name.

s size name ;
Creates a storage node with the specified name and positive integer size. Statements refer to an input or a storage node by its name or by '*n', where the positive integer 'n' specifies the nth node created by the network description.

m node1 node2 ;
Merges the two specified nodes together to create a single node. If both argument nodes are storage type, the resulting node is storage type and has size equal to the larger of the sizes of the argument nodes. Otherwise, the resulting node is input type.

n strength gate source drain ;
Creates an n-type transistor with the specified gate, source, and drain nodes and positive integer strength.

p strength gate source drain ;
Creates an p-type transistor with the specified gate, source, and drain nodes and positive integer strength.

d strength gate source drain ;
Creates an d-type transistor with the specified gate, source, and drain nodes and positive integer strength. Statements refer to a transistor by '?n', where the positive integer 'n' specifies the n-th transistor created by the network description.

v name { node | transistor } ;
Declares a vector called name to represent a sequence of nodes and transistors.

| { text };
Documents the network with the specified comment text.

AUTHOR

Michael Schuster

LIMITATIONS

The sum of the largest node size and the largest transistor strength must be strictly less then fifteen.

Node, vector, and faulty circuit names must not contain embedded colons ':' or vertical bars '|'.

Command and statement lines longer than 1023 characters are silently truncated.

BUGS

The **read** and **initialize** commands do not set the power and ground nodes Vdd and Gnd to 1 and 0, respectively.

The character '~' at the begining of a filename is not expanded to the home directory.

The commands 'force a:1 a|a@0:0' and 'force a|a@0:0 a:1' are not equivalent. The first command forces 'a' to 0 in faulty circuit 'a@0' and to 1 in the good circuit and all other faulty circuits. The second command forces 'a' to 1 in all circuits.

Appendix – Network Description Example

The text shown below is a switch-level description of the pseudo-static register shown in Figure 26. The first line is a comment that documents the circuit. The second, third, fourth, and fifth lines create the input nodes vdd, gnd, up, and down. The sixth creates the storage node bus with size k_2. The seventh line creates a n-type transistor with strength g_2 whose gate, source, and drain nodes are up, vdd, and bus, respectively. The rest of the circuit is described in the subsequent lines. The last line creates a vector named inputs which consists of the sequence of nodes up, down, read, write, and refresh.

```
| pseudo-static register circuit ;
i vdd ;
i gnd ;
i up ;
i down ;
s 2 bus ;
n 2 up vdd bus ;
n 2 down gnd bus ;
s 1 out ;
d 1 out out vdd ;
n 2 bus out gnd ;
i read ;
i write ;
i refresh ;
s 1 a ;
s 1 b ;
s 1 c ;
n 1 write bus a ;
n 1 refresh a c ;
n 2 read c bus ;
d 1 b b vdd ;
n 2 a b gnd ;
d 1 c c vdd ;
n 2 b gnd c ;
v inputs up down read write refresh ;
```

Appendix – Command File Example

The text shown below is a list of commands that will apply the test patterns in Figure 27 to the pseudo-static register circuit shown in Figure 26. The first command documents the simulation. The second command reads the network description of the circuit from the file pseudo.ntk. The third command sets the states of the nodes vdd and gnd to 1 and 0, respectively. The fourth command creates a faulty circuit for every possible node stuck-at-zero and node stuck-at-one fault. The fifth command drops the fault circuits containing stuck-at faults on the nodes vdd and gnd. The sixth command sets the hard detect limit to 1 so that a fault will be detected as soon as it produces a state on the node out that differs from the state produced by the good circuit with the requirement that neither state be X. The rest of the commands are grouped into sections each containing four commands. The first command of each group sets the inputs of the circuit to the states specified by the next test pattern. The second command performs unit simulation steps to propagate the effects of the test pattern throughout the circuit. The third command lists the state of the node out for the good circuit. The last command of each group checks the state of the node out for each faulty circuit to see if any faults have been detected. If a fault is detected, this command also discontinues the simulation of the corresponding faulty circuit. The last command terminates the simulation.

```
comment test patterns for the pseudo-static register circuit
read pseudo
set vdd:1 gnd:0
fault *@0 *@1
unfault vdd@0 vdd@1 gnd@0 gnd@1
option hard:1
set inputs:01101
simulate
get out
observe out
set inputs:10010
simulate
get out
observe out
set inputs:01001
simulate
get out
observe out
set inputs:01100
simulate
get out
observe out
set inputs:00000
simulate
get out
observe out
set inputs:00100
simulate
get out
observe out
set inputs:01101
simulate
get out
observe out
```

```
set inputs:00100
simulate
get out
observe out
quit
```